(Revised version December 23, 2015)

The C standard formalized in Coq

Proefschrift

ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus,
volgens besluit van het college van decanen
in het openbaar te verdedigen op dinsdag 1 december 2015
om 10:30 uur precies

door

Robbert Jan Krebbers

geboren op 22 september 1987
te Groesbeek

Promotor:

Prof. dr. Herman Geuvers

Copromotor:

Dr. Freek Wiedijk

Manuscriptcommissie:

Prof. dr. Frits Vaandrager (voorzitter)
Prof. dr. Gilles Barthe (IMDEA, Madrid, Spanje)
Prof. dr. Lars Birkedal (Aarhus University, Denemarken)
Dr. Xavier Leroy (Inria Paris-Rocquencourt, Frankrijk)
Dr. Michael Norrish (NICTA, Sydney, Australië)

Radboud University

This research has been carried out under the auspices of iCIS (institute for Computing and Information Science) of the Radboud University Nijmegen and the research school IPA (Institute for Programming research and Algorithmics).

NWO
Netherlands Organisation for Scientific Research

This research has been supported by NWO (grant number 612.001.014).

ISBN 978-94-6259-903-1
Typeset with LaTeX 2_{ε}
Printed and published by Ipskamp Drukkers Nijmegen

Publications

The majority of the results presented in this thesis have been published in the papers listed below. Although some papers are written with coauthors, they are primarily based on my own work. Apart from the results presented in this thesis, I have worked on several other topics before and during my PhD that have led to a number of refereed papers. The Coq repository [17] contains a formalization of all results in this thesis.

Refereed papers

[1] **Robbert Krebbers** and Freek Wiedijk. A typed C11 semantics for interactive theorem proving. In Xavier Leroy and Alwen Tiu, editors, *CPP*, pages 15–27, 2015. Chapters 2 and 7 are based on this paper.

[2] **Robbert Krebbers**. Separation Algebras for C Verification in Coq. In Dimitra Giannakopoulou and Daniel Kroening, editors, *VSTTE*, volume 8471 of *LNCS*, pages 150–166, 2014. Chapters 4 and 8 are based on this paper.

[3] **Robbert Krebbers**, Xavier Leroy, and Freek Wiedijk. Formal C semantics: CompCert and the C standard. In Gerwin Klein and Ruben Gamboa, editors, *ITP*, volume 8558 of *LNCS*, pages 543–548, 2014. Parts of Chapter 2 are based on this paper.

[4] **Robbert Krebbers**. An Operational and Axiomatic Semantics for Non-determinism and Sequence Points in C. In Suresh Jagannathan and Peter Sewell, editors, *POPL*, pages 101–112, 2014. Chapters 6 and 8 are based on this paper.

[5] **Robbert Krebbers**. Aliasing Restrictions of C11 Formalized in Coq. In Georges Gonthier and Michael Norrish, editors, *CPP*, volume 8307 of *LNCS*, pages 50–65, 2013. Chapters 3 and 5 are based on this paper.

[6] **Robbert Krebbers** and Freek Wiedijk. Separation Logic for Non-local Control Flow and Block Scope Variables. In Frank Pfenning, editor, *FoSSaCS*, volume 7794 of *LNCS*, pages 257–272, 2013. Chapters 6 and 8 are based on this paper. I have received an *“ETAPS best student contribution award”* for this paper.

[7] **Robbert Krebbers** and Freek Wiedijk. A Formalization of the C99 Standard in HOL, Isabelle and Coq. In James H. Davenport, William M. Farmer, Josef Urban, and Florian Rabe, editors, *Calculemus/MKM*, volume 6824 of *LNCS*, pages 301–303, 2011. Chapter 1 is based on this paper.

[8] **Robbert Krebbers**. A Formal C Memory Model for Separation Logic, 2015. Accepted with revisions for Journal of Automated Reasoning. Chapters 2–5 and Chapter 8 are based on this paper.

Non-refereed papers and drafts

[9] **Robbert Krebbers**, Louis Parlant, and Alexandra Silva. Moessner's Theorem: an exercise in coinductive reasoning in Coq, 2015. Submitted.

[10] **Robbert Krebbers** and Freek Wiedijk. Subtleties of the ISO C standard, 2012. Note. Chapter 2 is based on this paper.

Refereed papers not part of this thesis

[11] **Robbert Krebbers** and Bas Spitters. Type classes for efficient exact real arithmetic in Coq. *Logical Methods in Computer Science*, 9(1):1–27, 2013.

[12] **Robbert Krebbers**. A call-by-value lambda-calculus with lists and control. In Herman Geuvers and Ugo de'Liguoro, editors, *CL&C*, volume 97 of *EPTCS*, pages 19–33, 2012.

[13] Herman Geuvers, **Robbert Krebbers**, and James McKinna. The $\lambda\mu^{\mathbf{T}}$-calculus. *Annals of Pure Applied Logic*, 164(6):676–701, 2013.

[14] **Robbert Krebbers** and Bas Spitters. Computer Certified Efficient Exact Reals in Coq. In James H. Davenport, William M. Farmer, Josef Urban, and Florian Rabe, editors, *Calculemus/MKM*, volume 6824 of *LNCS*, pages 90–106, 2011.

[15] Herman Geuvers and **Robbert Krebbers**. The correctness of Newman's typability algorithm and some of its extensions. *Theoretical Computer Science*, 412(28):3242–3261, 2011.

[16] Herman Geuvers, **Robbert Krebbers**, James McKinna, and Freek Wiedijk. Pure Type Systems without Explicit Contexts. In Karl Crary and Marino Miculan, editors, *LFMTP*, volume 34 of *EPTCS*, pages 53–67, 2010.

Notable Coq formalizations

[17] **Robbert Krebbers** and Freek Wiedijk. The CH_2O formalization, 2015. Available online at `http://robbertkrebbers.nl/research/ch2o/`.

[18] **Robbert Krebbers**, Bas Spitters, and Eelis van der Weegen. Math classes: a library of abstract interfaces for mathematical structures, 2015. Available online at `https://github.com/math-classes/math-classes`.

[19] Henk Barendregt, Luís Cruz-Filipe, Herman Geuvers, Mariusz Giero, Rik van Ginneken, Dimitri Hendriks, Jelle Herold, Sébastien Hinderer, Cezary Kaliszyk, Bart Kirkels, **Robbert Krebbers**, Pierre Letouzey, Iris Loeb, Evgeny Makarov, Lionel Mamane, Milad Niqui, Russell O'Connor, Randy Pollack, Nickolay Shmyrev, Bas Spitters, Dan Synek, Eelis van der Weegen, Freek Wiedijk, and Jan Zwanenburg. CORN: the Coq Repository at Nijmegen, 2015. Available online at `http://corn.cs.ru.nl/`.

Contents

Acknowledgments

First and foremost, I would like to thank Freek Wiedijk for his supervision and continuous support during my PhD. His capability of being able to understand complicated subjects instantly never ceases to amaze me. I have enjoyed that I could always enter his office to discuss anything. I appreciate that he has always supported me to take my own research direction, whether he agreed on that direction or not.

I would like to thank my promotor, Herman Geuvers. Before my PhD, Herman has supervised me on numerous projects, including my bachelor's and master's thesis. Working with Herman has always been a pleasure, and pursuing a PhD under his supervision was therefore the obvious next step. His enthusiasm and ability to explain difficult topics in a clear matter have always been valuable.

This thesis has been reviewed by a manuscript consisting of five people. I would like to thank Frits Vaandrager, Gilles Barthe, Lars Birkedal, Xavier Leroy and Michael Norrish for taking the time to review my manuscript. When I will defend this thesis, it will be before a committee consiting of Henk Barendregt, Yves Bertot, Lars Birkedal, Marko van Eekelen, Xavier Leroy, Erik Poll, Peter Schwabe and Frits Vaandrager. I would to thank them for being there.

I have published various papers during my PhD. I would like to thank my co-authors Alexandra, Bas, Freek, Herman, James, Louis and Xavier for their collaboration and the anonymous reviewers for their feedback.

The direction of this thesis has been strongly influenced by Erik Poll who suggested me to look into an axiomatic semantics for non-local control. This has resulted in my FoSSaCS'13 paper, for which I have received an ETAPS best student contribution award. Erik, without your suggestion this would not have happened and this thesis would not have been the way it is.

I would like to thank James McKinna for (co-)supervising my master thesis and various other master projects, as well as providing valuable feedback on this thesis. His knowledge of science, the English language, politics and many other things has made me a better scientist.

Before I started my Phd, I worked with Bas Spitters on the formalization of real numbers. Although we have worked on it for only a short period of time, that period was very productive due to his enthusiasm and knowledge.

I am indebted to Xavier Leroy for many useful discussions on the semantics of C, the internals of CompCert and formalization in general. He always took the effort to answer my questions is the most comprehensive way possible.

During my PhD I have been lucky to meet many fantastic researchers. In particular, I would like to thank Xavier Leroy for making it possible to visit his group in Paris and Andrew Appel for making it possible to visit his group in Princeton. Moreover, I would like to thank the organizers of the Oregon Programming Languages Summer School as well as the organizers of the IHP thematic trimester, where I had the pleasure to meet a lot of nice people.

I would like to thank the Coq team for the development of the Coq proof assistant. Coq has played a central role in my research.

Thank you Henning and Jonce for being my paranimphs. I also would like to thank Henning for his help while writing this thesis, in particular for answering tricky questions about LaTeX.

During my PhD I have been involved in quite some teaching. I would like to thank Freek and Herman for teaching type theory with me, and Engelbert for being the most dedicated and well organized teacher a teaching assistant could wish for.

I have had the pleasure to be part of the Intelligent Systems group of the Radboud University. I would like the members (as well as frequent visitors and ex-members) of the group: Alexandra, Ali, Andrew, Bas J., Bram, Carst, Daniel, Daniela, Darya, Dexter, Eelis, Elena M., Elena S., Fabian, Fabio, Freek, Georgiana, Giulio, Hans, Helle, Henk, Henning, Herman, Jacopo, James, Jan, Jonce, Josef, Kasper, Lionel, Matteo, Maya, Mark, Max, Nicole, Ridho, Saiden, Saskia, Simone M., Simone L., Suzan, Thijs, Tom C., Tom H., Tameem, Twan, Wout and Wouter. Apart from those in the Inteligent Systems group, I have enjoyed contacts with others of the computer science department. Although I will not be able to thank all of those personally, I would like to thank Bas W., Freek V., Joshua, Peter and Robert.

Last, but not least, I would like to thank my family and friends for their support. In particular, thank you Irene, Jan, Annelies, Lennert and Vivian.

Robbert Krebbers
Aarhus, October 25, 2015

Chapter 1

Introduction

The C programming language was created by Thompson and Ritchie around 1970 as the implementation language of the Unix operating system [Rit93]. The development of Unix demonstrated the efficiency and portability of C, and following that success, C quickly became a dominant general purpose programming language.

More than 40 years after its introduction, C remains among the most widely used programming languages in the world (see the TIOBE [TIO] and IEEE [IEE15] indices). However, despite its continuing wide use, C is also among the most bug-prone programming languages in the world. As a result of weak static typing and the absence of run-time checks, it is very easy for C programs to have bugs that make the program crash or behave badly in other ways. Dangling pointers and `NULL` pointers can be dereferenced, arrays can be accessed outside their bounds, *etc.*

A recent example is the Heartbleed bug in the widely used OpenSSL cryptography library where a buffer overflow allowed access to arbitrary data, which may contain passwords [MIT, CVE-2014-0160]. Heartbleed is not an incidental case where the unsafety of C has disastrous consequences. Wang *et al.* [WCC$^+$12] have shown that the unsafety of C is a serious problem. In safer programming languages than C, bugs like these are less likely to occur, but due to the performance, control and portability benefits of C, the use of C and C derivatives like C++ remains widespread.

Formal verification is a promising approach to retain the performance, control and portability benefits of C but without the dangers of its unsafety. In formal verification one uses mathematical methods to obtain the highest level of assurance of a program's safety, or even of its entire functional correctness.

In order to advance the use of formal verification applied to C, this thesis describes a formal semantics corresponding to a significant part of the official C language specification, the C11 standard [ISO12], as well as technology to enable verification of C programs in a standards compliant and compiler independent way.

1.1 The C standard

The tutorial book "The C Programming Language" by Kernighan and Ritchie, as published in 1978 [KR78], has been considered the *de facto* language specification of

C for over a decade. But as C gained popularity, there became an increasing need for a definite language specification to keep C dialects from diverging [ISO03].

In 1983, the American National Standards Institute (ANSI) formed a committee to standardize C, which resulted in the C89 standard [ANS89]. The C89 standard was later adopted by the International Organization for Standardization (ISO) and issued as the C90 standard.

The C99 standard [ISO99] has extended C89/C90 with many features (for example, intermingled code and variable declarations, **`long long`**, compound literals, effective types, variable length arrays, *etc.*), contains reworked descriptions of certain features, and corrections of many mistakes. The most recent version, the C11 standard [ISO12], has continued reworking descriptions and correcting mistakes, but is most notable for its added support of concurrency.

Since the standard is intended to be the definite language specification, its main goal is to be clear, consistent, and unambiguous [ISO03]. However, just like any text written in prose style natural language, this goal often remains to be desired. This is particularly troublesome in the following cases:

- The C standard is a contract between compiler writers and programmers. Misunderstanding of the standard text by compiler writers results in too aggressive optimizations or other forms of miscompilation. Misunderstanding of the standard text by programmers results in programming bugs.
- In order to apply formal verification to C, a mathematically precise language specification is needed. In formal mathematics one cannot cut corners like prose style texts such as the C standard are able to do.

The fact that the C standard indeed suffers from ambiguities is witnessed by the numerous requests for clarification in the form of defect reports and the numerous discussions on the standard committee's mailing list [ISO].

Many obscurities of C11 are related to the interaction between *low-level* and *high-level* data access in C. Low-level data access involves unstructured and untyped byte representations whereas high-level data access involves abstract values such as structs and unions. Compilers often use a high-level view of data access to perform optimizations whereas programmers expect data access to behave in a low-level way.

This interaction has led to numerous ambiguities in the standard text related to aliasing, uninitialized memory, end-of-array pointers and type-punning. See for example the message [Mac01] on the standard committee's mailing list, Defect Reports #236, #260, and #451 [ISO], and the various examples in this thesis.

1.2 Formal C verification

Formal verification is the process of using mathematical methods to prove properties of programs. There is a wide range of approaches towards formal verification. Some require human guidance and guarantee full functional correctness, whereas others are fully automated and merely guarantee the absence of run-time errors (such as out of bounds array accesses) or are just meant to find bugs.

The basic concepts of verification have been pioneered by Floyd and Hoare in the 1960s [Flo67, Hoa69]. They presented an axiomatic system, nowadays called *a Hoare* *logic*, which allows reasoning about programs in a structured fashion. Building on Floyd and Hoare, Dijkstra [Dij75] invented the concept of *weakest preconditions*. This concept forms the basis of verification condition generation, which given a program with logical annotations, produces a set of verification conditions. If all generated conditions are proven, the program is guaranteed to be correct.

Another approach to formal verification is *abstract interpretation*, which has been invented by Cousot and Cousot in the 1970s [CC77]. The idea of abstract interpretation is to compute an over-approximation of a given program's behavior. Properties are then proven with respect to the over-approximation and related back to the concrete behavior of the program.

Abstract interpretation is the key to static program analysis, which is widely used to fully automatically prove shape and safety related properties of programs. Examples of such systems are the static analyzer of Clang, the Sparse static analyzer, which is used for the Linux kernel among others, the Coverity static analyzer [BBC$^+$10], which is used for software in the Large Hadron Collider and the Mars rover Curiosity among others, and the Astrée static analyzer [BCC$^+$03] used by Airbus among others. Systems that are based on combinations of static analysis and model checking have also proven successful, for example Microsoft's SLAM static driver verifier [BLR11] and the Berkeley Lazy Abstraction Software Verification Tool [BHJM07].

Since systems for static analysis operate on an approximation of the behavior of the given program, they are inherently incomplete. In practice that means these systems will either yield false-negatives from time to time or have to sacrifice soundness. The emphasis of most systems for static analysis is thus on bug finding.

Human guided systems reduce the problem of incompleteness, and are therefore more general. These systems are mostly based on a form of verification condition generation where humans guidance is required to annotate the given program with logical conditions. The resulting verification conditions are then verified either fully automatically using SAT/SMT solvers, or interactively using *proof assistants*. Examples of such systems are Boogie [BMSW10], HAVOC [BHL$^+$10], the Jessie plug-in [MM11] of Frama-C [CKK$^+$12], Key-C [MLH07], VCC [CDH$^+$09] and Verifast [JP08]. These systems have for example been used for the verification of a microkernel operating system written in C [CDH$^+$09].

A shortcoming of the aforementioned systems is the use of an *implicit* semantics of C. That means, the semantics of C is part of the implementation of the system and has not been stated explicitly. The semantics thus cannot be studied on its own, and is very difficult to get correct with respect to the C standard. Indeed, as shown in [ER12a, Ell12], many of these systems fail to get correct subtle behaviors of C such as unspecified order of expression evaluation.

In order to obtain the best environment to reason reliably about C programs, one needs an *explicit* semantics in a proof assistant such as Coq [Coq15], Isabelle [WPN08], or HOL4 [SN08]. The proof assistant can then be used as a unified environment to establish metatheoretical properties of the language, to verify actual C programs, and to verify the correctness of C compilers and static analyzers.

The first significant formalized C semantics has been developed by Norrish who used HOL4 to formalize delicate parts of the C89 standard [Nor99]. A few years later, Leroy achieved a milestone in compiler verification. He has formalized a large part of the C semantics in Coq which he has used to prove the correctness of an optimizing compiler, called CompCert [Ler06]. CompCert is written in Coq itself. Notable uses of the CompCert semantics include the verification in Coq of an implementation of SHA [App15], a static analyzer for C [JLB$^+$15], and a microkernel operating system written in C [GVF$^+$11].

An entire different approach to C verification is the L4.verified project by Klein *et al.* [K$^+$09] where the C sources of a microkernel operating system are translated into monadic definitions inside the Isabelle proof assistant [GLAK14]. Verification is then performed with respect to these monadic definitions. The C sources are finally compiled using GCC, and the produced assembly is related to the Isabelle definitions using translation validation [SMK13].

Both the CompCert and L4.verified project use a specific semantics of C. That means, they use a semantics that is tied to a specific compiler or computing architecture and thus do not capture all behaviors allowed by the C standard. Verification of a program against the CompCert semantics gives very strong guarantees when the program is compiled with the CompCert compiler but does not give reliable guarantees when the program is compiled using a different compiler.

Since these projects are aimed towards C verification involving a specific computing architecture or compiler, their choice for using a semantics that describes fewer behaviors than the C11 standard is appropriate. However, in order to apply C verification to programs compiled with arbitrary C11 compliant compilers, one needs a C semantics that captures all behaviors allowed by the C11 standard.

Ellison and Roşu [ER12b] have developed such a semantics of C11 using the $\mathbb{K}$ rewriting framework. Their semantics covers an impressive fragment of C close to C11 and is parameterized by implementation choices. Their semantics is primarily aimed at being executable and has been used as an oracle for compiler testing [RCC$^+$12]. However, their semantics is unlikely to be of practical use in proof assistants because it is defined on top of a large C abstract syntax and uses a rather ad-hoc execution state that contains over 90 components.

In summary, the last decades have shown a diversity of impressive results in formal verification applied to real life languages like C. In order to advance C verification and to make it more reliable, this thesis presents an explicit semantics of C formalized in Coq. Such a semantics is an important artifact and has four main applications:

- It makes the standard utterly precise. This is useful for compiler writers, who will get the means to establish how the standard needs to be understood without having to deal with the ambiguities of prose style natural language. Programmers writing C code get the same benefit.
- It allows one to prove metatheoretical properties of the language specification. These properties are essential to validate the formal definitions.
- When establishing a property of a C program, it is very attractive to be able to claim that the property has been proven with respect to the official standard and thus conforms to widely used compilers such as GCC and Clang. This kind of knowledge is implicit in most current tools. A formal semantics of C11 makes it possible to make this connection with the standard explicit.

- Verified compilers such as CompCert [Ler09a] need a semantics of a version of C. With a formal version of the C11 semantics, the correctness of the compiler becomes provable with respect to the official standard.

In order to make these applications possible, our semantics captures all behaviors allowed by C11, supports a substantial part of the C language, and is usable in proof assistants. All our results are formalized in the Coq proof assistant.

1.3 The CH_2O project

This thesis describes the results of the CH_2O project. The goal of the CH_2O project is to develop a formal[1] version of the non-concurrent fragment of the C11 standard that is usable in proof assistants. The CH_2O project provides the following kinds of formal semantics for C, along with proofs that they correspond to each other.

- **Operational semantics.** An operational semantics describes the behavior of programs using individual computational steps. It is generally used to reason about program transformations and to prove metatheoretical properties.
- **Executable semantics.** An executable semantics is an algorithmic version of the operational semantics that allows one to *compute* the set of behaviors of a given program. It is generally used for debugging and testing purposes.
- **Axiomatic semantics.** An axiomatic semantics allows one to reason about programs in a structured fashion. It is generally used to reason about concrete C programs.

Given the different purposes of these kinds of formal semantics, it is important to have corresponding versions of these kinds. Without a corresponding executable semantics it is difficult to test whether the semantics gives the intended behavior to example programs, and without an axiomatic semantics it is difficult to apply it to program verification.

The CH_2O project does not just consider an academic fragment of the C language. It considers a significant part that includes many delicate features:

- Integer types, including `long long`, `size_t`, `ptrdiff_t`, *etc.*
- Implicit type conversions (usual arithmetic conversions and integer promotions).
- Pointer types, including function pointers and `void*` pointers. Pointers can be end-of-array and be cast to `unsigned char*` to access object representations.
- Array types, struct and union types (also as argument and return types of functions), enum types and typedefs.
- Expressions with side-effects such as assignment (operators), function calls and the sequencing operators `(_ ? _ : _)`, `(_ , _)`, `(_ && _)` and `(_ || _)`.
- Loops (`while`, `for` and `do-while`), basic control (`if-else`), non-local control (`goto`, `break`, `continue` and `return`) and the `switch` statement (including unstructured variants such as Duff's device).

[1] A solution of the chemical compound CH_2O in water is called formalin.

- Scoping, including local variables with **`static`** and **`extern`** storage specifiers.
- Dynamically allocated storage through `malloc` and `free`.
- Initializers, compound literals and constant expressions.
- Implementation-defined operators and constants such as **`sizeof`**, `_Alignof`, `offsetof`, `CHAR_BIT`, `INT_MIN`, `INT_MAX`, *etc.*

During the development of CH_2O we have kept the following requirements and principles in mind:

- **Close to C11.** CH_2O is faithful to the C11 standard in order to be compiler independent. When one proves something about a given program with respect to CH_2O, it should behave that way with *any* C11 compliant compiler (possibly restricted to certain implementation-defined choices).
- **Static type system.** Given that C is a statically typed language, CH_2O does not only capture the dynamic semantics of C11 but also its type system. We have established properties such as type preservation and a limited form of progress.
- **Proof infrastructure.** All parts of the CH_2O project with the exception of the parser have been formalized in Coq (without axioms). This is essential for its application to program verification in proof assistants. Also, considering the significant size of CH_2O, proving metatheoretical properties of the language would have been intractable without the support of a proof assistant.

  Despite our choice to use Coq, we believe that nearly all parts of CH_2O could be formalized in any proof assistant based on higher-order logic.
- **Based on a core language.** The definition of C abstract syntax contains a lot of delicate details such as constant expressions and initializers, and consists of a large body of mutually dependent parts. Defining a semantics directly on top of C abstract syntax would make it unusably complicated.

  The three versions of the CH_2O semantics are defined on top of a more principled language called **CH_2O core C**. Our language **CH_2O abstract C**, which is very close to C abstract syntax, is given a semantics by a translation into CH_2O core C. This translation is defined in Coq and proven to be type sound.
- **Memory refinements.** CH_2O has an expressive notion of memory refinements that relates memory states. The operational semantics is proven invariant under this notion. Memory refinements form a general way to validate many commonsense properties of the memory model in a formal way. They also open the door to reasoning about program transformations.
- **Separation logic.** Separation logic is a modern variant of Hoare logic that allows one to reason about imperative programs that use mutable data structures and pointers [ORY01]. CH_2O has an axiomatic semantics in the form of a separation logic.

1.4 Overview of this thesis

The structure of this thesis is bottom up: we will define the various C semantics starting from small subcomponents towards the entire semantics. This section gives

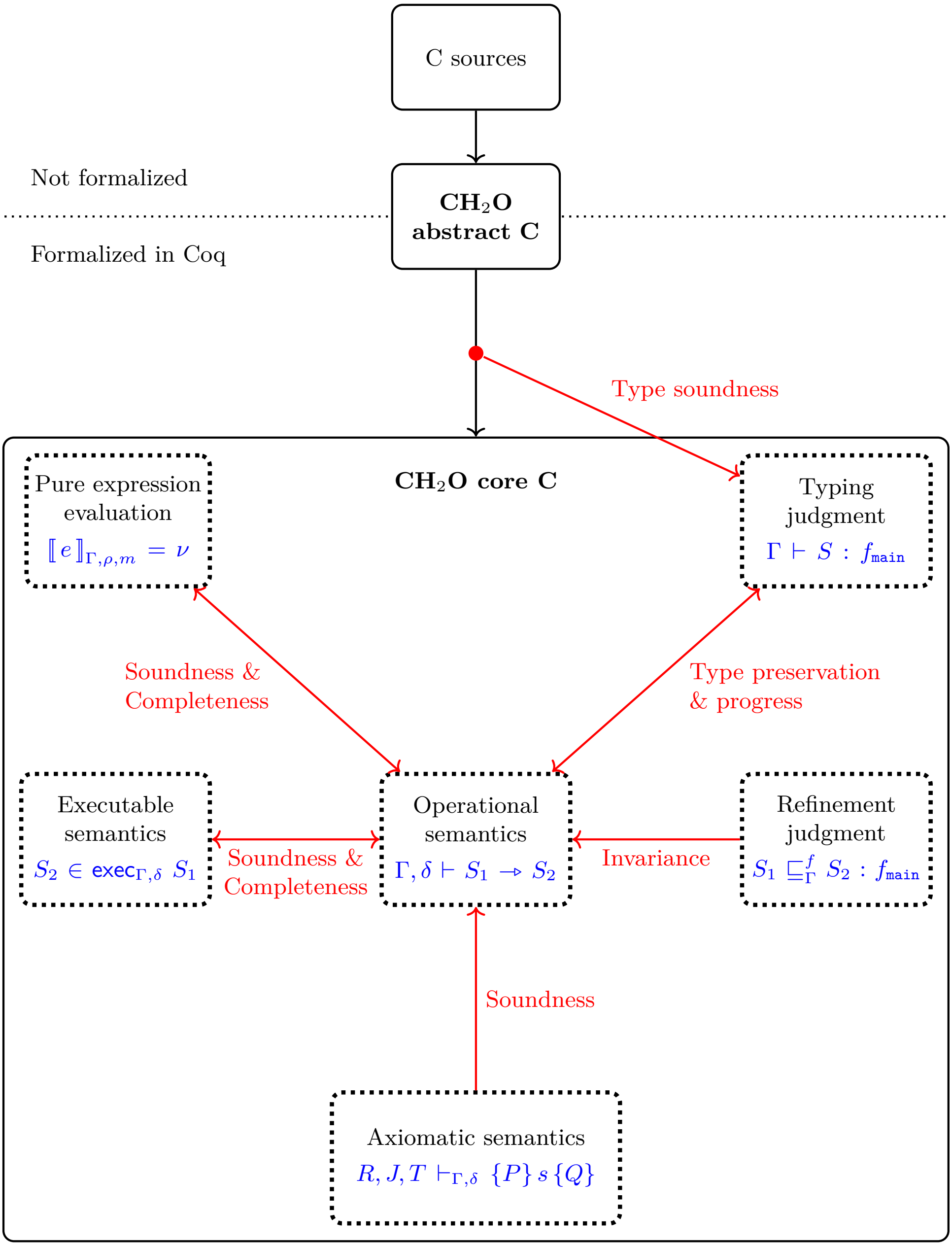
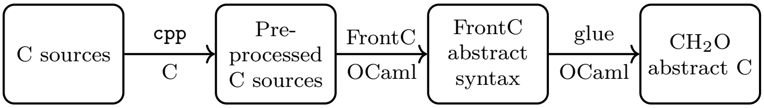

C sources
Not formalized
Formalized in Coq
$\mathbf{CH_2O}$ abstract C
Type soundness
$\mathbf{CH_2O}$ core C
Pure expression evaluation
$[\![e]\!]_{\Gamma,\rho,m} = \nu$
Typing judgment
$\Gamma \vdash S : f_{\mathtt{main}}$
Soundness & Completeness
Type preservation & progress
Executable semantics
$S_2 \in \mathsf{exec}_{\Gamma,\delta}\ S_1$
Soundness & Completeness
Operational semantics
$\Gamma, \delta \vdash S_1 \rightharpoonup S_2$
Invariance
Refinement judgment
$S_1 \sqsubseteq^{f}_{\Gamma} S_2 : f_{\mathtt{main}}$
Soundness
Axiomatic semantics
$R, J, T \vdash_{\Gamma,\delta} \{P\}\, s\, \{Q\}$

Figure 1.1: Overview of the components of the CH_2O project. Boxes denote languages and arrows marked black denote translations between these languages. Dotted boxes denote judgments (such as a semantics) and arrows marked red denote important proofs related to these judgments.

a top-down overview of the thesis as graphically displayed in Figure 1.1. In this thesis we consider two formalized languages:

- **CH_2O core C**. The syntax of this language resembles actual C but incorporates many simplifications to make its semantics more principled. For example, CH_2O core C has just one looping construct that generalizes the various looping constructs of C (**`while`**, **`for`** and **`do-while`** loops), uses De Bruijn indices for local variables, and makes l-value conversion explicit.
- **CH_2O abstract C**. This language bridges the gap between CH_2O core C and C abstract syntax. It uses named variables, supports global and static variables, initializers, enum types, typedefs, *etc.* The semantics of CH_2O abstract C is specified by translation into CH_2O core C.

We use a small part that is not formalized to translate actual C source files into CH_2O abstract C. This translation consists of the following phases:



The GNU C preprocessor (called `cpp` in the diagram) is used to perform macro expansion and inclusion of header files. The FrontC parser[2] is used to transform the preprocessed C sources into an OCaml representation of an abstract syntax tree. This abstract syntax tree is then translated into CH_2O abstract C using a thin layer of OCaml glue.

The languages CH_2O core C and CH_2O abstract C as well as their translation are formalized in Coq. The translation is proven to be type sound. Type soundness gives a higher confidence in the correctness of the translation. The various C semantics that we present will be defined on top of CH_2O core C.

Operational semantics. Computation in the small-step operational semantics is defined as the reflexive transitive closure of a reduction relation $\Gamma, \delta \vdash S_1 \to S_2$. The environments Γ and δ serve the following purposes:

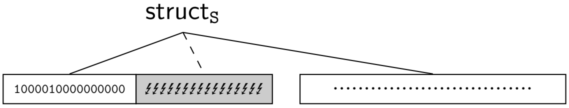
- The environment $\Gamma \in \mathsf{env}$ assigns field types to struct and union types as well as argument and return types to function names.
- The environment $\delta \in \mathsf{funenv}$ contains the bodies of all declared functions.

States $S \in \mathsf{state}$ consist of two components: the memory and the current location in the program that is being executed. The memory corresponding to the initial state contains storage for global and static variables.

Memories $m \in \mathsf{mem}$ are represented as forests of well-typed trees whose structure corresponds to the structure of data types in C. The tree corresponding to the

[2] Our version of FrontC is based on the version used by CompCert version 2.2 [Ler09a], which in turn is based on the FrontC parser of the CIL suite [NMRW02]. Note that FrontC is just a lexer and parser. Contrary to the full CIL suite, it does not perform any code transformations to the C abstract syntax trees such as pulling out side-effects of expressions.

declaration **`struct`** `S {` **`short`** `x, *r; } s = { 33, &s.x }` might be (the precise shape and the bit representations are implementation-defined):



The leaves of these trees contain symbolic bits: the integer `33` is represented as its binary representation `1000010000000000`, the padding as symbolic *indeterminate* bits ↯ (whose actual values should not be used), and the pointer `&s.x` as a sequence of symbolic *pointer bits*. Bits are annotated with permissions (not displayed).

Our structured memory model gives an unambiguous semantics to delicate features of C that have not yet been addressed by others. In particular, the tree structure naturally models the standard's notion of *effective types* [ISO12, 6.5p6-7] which allow compilers to perform optimizations based on type-based alias analysis.

The current location in the program that is being executed is described using an adaptation of Huet's zipper data structure [Hue97] over statement abstract syntax trees. We use the zipper data structure to accurately describe non-local control (**`goto`**, **`break`**, **`continue`**, **`return`** and unstructured **`switch`**) in the presence of block scope local variables. The current location in the program is a pair $(\mathcal{P}, \phi)$ where $\mathcal{P} \in \mathsf{ctx}$ is the context of the part $\phi \in \mathsf{focus}$ that is being executed. We consider the following forms of program execution:

- **Execution of a statement.** Execution of a statement occurs by small-step traversal through the zipper in a direction corresponding to the current form of (non-local) control. The context $\mathcal{P}$ of our zipper implicitly contains the program stack $\rho \in \mathsf{stack}$ that assigns locations in memory to local variables.
- **Execution of an expression.** Execution of an expression is described by a head reduction $\Gamma, \rho \vdash (e_1, m_1) \multimap_{\mathsf{h}} (e_2, m_2)$. Evaluation contexts [FFKD87] are used to select a head redex in a whole expression.
- **Calling a function** and **returning from a function.** When calling a function, the zipper is extended with the function body of the callee, which is in turn executed. Returning from a function resumes execution of the caller, whose location is stored as part of the context $\mathcal{P}$ of our zipper.
- **Undefined behavior.** We use a special $\overline{\mathsf{undef}}$ state to describe that a run-time error (undefined behavior in C terminology) has occurred.

Typing judgment. The typing judgment for states has the shape $\Gamma \vdash S : f_{\mathtt{main}}$ where $f_{\mathtt{main}} \in \mathsf{funname}$ is the function that started the computation that led to S. We have proven type preservation (reduction in the operational semantics preserves typing) and a limited form of progress (every non-final non-error state can reduce).

Refinement judgment. The refinement judgment $S_1 \sqsubseteq_\Gamma^f S_2 : f_{\mathtt{main}}$ allows one to relate states with each other. The state S_2 may be more defined than S_1 (*i.e.* it may have fewer indeterminate memory locations). The function f is used to relabel the memory and to coalesce multiple top-level objects into subobjects of a larger object. The operational semantics is proven invariant under refinements in the form of a backward simulation.

Executable semantics. The executable semantics $S_2 \in \mathsf{exec}_{\Gamma,\delta}\ S_1$ is an algorithmic version of the operational semantics. That means, the function $\mathsf{exec}_{\Gamma,\delta} : \mathsf{state} \to \mathcal{P}_{\mathsf{fin}}(\mathsf{state})$ *computes* the finite set of subsequent states.

The executable semantics is proven sound and complete with respect to the operational semantics. The completeness proof is complicated due to excessive nondeterminism in our operational semantics.

Pure expression evaluation. Pure expressions do not contain assignments and function calls and therefore enjoy three useful properties: they do not modify the memory state, they yield unique results, and their executions always terminate. These properties allow us to define an evaluator $[\![\_]\!]_{\Gamma,\rho,m} : \mathsf{expr} \to \mathsf{option}\ \mathsf{lrval}$ that yields the resulting address or abstract value of an expression.

The evaluator of pure expressions is used for constant expression evaluation during the translation of $\mathrm{CH_2O}$ abstract C into $\mathrm{CH_2O}$ core C and in assertions of separation logic. It is proven sound and complete with respect to the operational semantics.

Axiomatic semantics. Judgments of the axiomatic semantics are of the shape $R, J, T \vdash_{\Gamma,\delta} \{P\}\, s\, \{Q\}$. The right hand side $\{P\}\, s\, \{Q\}$ is a traditional Hoare triple. That means, P and Q are *assertions* called the pre- and postcondition. Provided P holds for the state before executing s, and execution of s terminates, then Q holds for the state after execution. On top of that, the axiomatic judgment ensures the absence of run-time errors (*undefined behavior* in C terminology). The environments deal with non-local control:

- The environment R specifies the conditions for returns.
- The environment J specifies the conditions for gotos.
- The environment T specifies the conditions for break and continue.

The axiomatic semantics is based on separation logic [ORY01] and is defined using a shallow embedding in terms of the operational semantics.

1.5 Is $\mathrm{CH_2O}$ really a formalization of C11?

The skeptic reader may wonder whether the definitions in this thesis really constitute a formalization of the C11 standard. In this section we will argue that $\mathrm{CH_2O}$ is indeed a correct formal definition of C.

- We have carefully studied the standard text, defect reports, and the behaviors of widely used C compilers. The formal definitions in this thesis are motivated by references to the standard text, and by contemplating example programs.
- Using our executable semantics we have debugged the semantics on example C programs. We have tested the semantics on all examples in this thesis, examples from the standard text and defect reports, as well as on a small test suite. This

shows that the set of possible behaviors (including *undefined behaviors*) that CH_2O assigns to these programs conforms with the C11 standard.

- We have designed CH_2O to be compatible with the formally verified CompCert C compiler by Leroy *et al.* [Ler09b]. This shows that our semantics does not treat too many programs as semantically illegal (having *undefined behavior*).

Since the C11 standard is written in English instead of a mathematically precise formalism, it is impossible to obtain an exact correspondence between CH_2O and the standard text. Even worse, as we will show in Chapter 2, we have found conflicts in the standard text that would make a mathematical formalization inconsistent. This has necessitated us to make specific design choices. We have made these design choices by treating doubtful programs as semantically illegal (having *undefined behavior*) so as to ensure that CH_2O is safe with respect to existing compilers. Chapter 2 contains an extensive discussion of this issue.

In order to verify that our choices are reasonable and unambiguous, we have made use of the following forms of validation:

- All our results have been formalized using the Coq proof assistant. Although a proof assistant does not guarantee that the definitions have the intended meaning, it provides a good sanity check as it requires one to carefully craft the definitions and it guarantees that these are mathematically well-formed.
- We have developed various metatheoretical results about the semantics. These results show that the semantics is well-behaved and enjoys intended properties. For example, it justifies important compiler optimizations.
- We have developed three different kinds of semantics for C along with proofs that they correspond to each other. We have thus considered C from multiple points of view, and thereby ensured that our definitions match our intent.

One may also wonder whether Coq gives higher confidence that our proofs are indeed correct. For this issue we refer to the extensive discussion on the trustworthiness of proof assistants by Pollack [Pol98].

1.6 Contributions

This thesis includes a number of contributions that have been published before. These contributions are useful in their own right, and may apply to other low-level programming languages such as C++ or LLVM.

- In order to accurately capture allocation and deallocation of block scope variables due to uses of `goto` or `return` statements, we have developed an approach for handling **non-local control flow in the presence of block scope local variables** [KW13]. Our approach uses an operational semantics in which the program state is represented through an adaptation of Huet's zipper data structure. We have proven soundness of a corresponding separation logic. I have received an "*ETAPS best student contribution award*" for this contribution.
- In order to handle non-determinism in expressions with side-effects we have developed an approach for **non-determinism and sequence points** [Kre14a].

Our approach uses a permission system to rule out multiple modifications of the same memory area in a single expression (which is undefined in C). We have proven soundness of a corresponding separation logic.

- We have developed a **typed memory model based on trees** [Kre13]. This memory model supports both low-level access to byte-level representations and high-level compiler optimizations based on abstract values (struct and unions). Our memory model gives an unambiguous semantics to delicate features of C such as effective types that have not yet been addressed by others.
- We have developed **a generalization of separation algebras that is well-suited for C verification** [Kre14b]. We have used our generalization of separation algebras to compositionally define the CH_2O permission system as a telescope of separation algebras. These separation algebras have also been used to make the CH_2O memory model suitable for separation logic.
- We have developed an **interpreter that calculates all behaviors of a given C program** with respect to our semantics [KW15]. This interpreter formally translates C abstract syntax into a core language, on top of which we have defined an executable semantics that computes the set of all behaviors allowed by C11. The executable semantics has been proven sound and complete with respect to the operational semantics. The translation into a core language extends the language with many delicate features of C such as initializers, compound literals, constant expressions and typedefs.
- We have extended **CompCert with some subtle behaviors of C** [KLW14], namely end-of-array pointers and the possibility to byte-wise copy objects. This extension brings CompCert closer to C11 and the CH_2O semantics.
- We have developed a **Coq support library**. This library has a great number of definitions and theory on common data structures such as lists, finite sets, finite maps and hashsets. Our Coq library uses type classes to overload common notations, such as those used when programming with monads.

1.7 Outline

This thesis consists of the following chapters:

Chapter 2 describes subtleties of C and our treatment of these. These subtleties are presented in the form of numerous example programs. This chapter furthermore gives a brief introduction to C. This chapter is based on [KW15] published in CPP'15, [KLW14] published in ITP'14 and [Kre15] accepted with revisions for JAR.

Chapter 3 describes our formal treatment of implementation-defined aspects of C such as integer representations, the behavior of integer operations, and the layout of structs and unions. This leads to a formal treatment of types in C. This chapter is based on [Kre13] published in CPP'13 and [Kre15] accepted with revisions for JAR.

Chapter 4 describes the CH_2O permission system which is built compositionally using separation algebras. This chapter is based on [Kre14b] published in VSTTE'14 and [Kre15] accepted with revisions for JAR.

Chapter 5 describes the CH_2O memory model. Our memory model is defined as a forest of trees with bits on their leaves. The combination of trees and bits gives an appropriate semantics to low-level operations as well as delicate high-level aspects of C11 such as effective types. This chapter establishes metatheoretical properties of the memory model related to common compiler optimizations. This chapter is based on [Kre13] published in CPP'13 and [Kre15] accepted with revisions for JAR.

Chapter 6 describes the language CH_2O core C and its operational and executable semantics. This chapter establishes desirable metatheoretical properties such as type preservation, progress, and soundness and completeness of both the executable semantics and the pure expression evaluator. This chapter is based on [KW13] published in FoSSaCS'13, [Kre14a] published in POPL'14 and [Kre15] accepted with revisions for JAR.

Chapter 7 describes the language CH_2O abstract C and its translation into CH_2O core C. This translation is proven type sound, and is combined with the executable semantics to obtain an interpreter that computes all behaviors of a C source file. We have used the interpreter to validate our semantics against all examples in this thesis and a small test suite. This chapter is based on [KW15] published in CPP'15.

Chapter 8 describes a separation logic for CH_2O core C. This separation logic supports novel features such as block scope local variables in the presence of non-local control and expressions with side-effects in the presence of non-deterministic expression evaluation. This chapter is based on [KW13] published in FoSSaCS'13, [Kre14a] published in POPL'14, [Kre14b] published in VSTTE'14 and [Kre15] accepted with revisions for JAR.

Chapter 9 describes the Coq formalization. All definitions and proofs in this thesis have been fully formalized using Coq. This chapter also describes a Coq support library developed as part of CH_2O.

Chapter 10 describes related work and **Chapter 11** finishes with conclusions and future work.

1.8 Coq sources

All definitions and proofs in this thesis have been formalized using Coq. The sources of the formalization are available online under the terms of a BSD license:

`http://robbertkrebbers.nl/research/ch2o`

Since this thesis describes a large formalization, we often omit details and proofs. The interested reader can find all details online as part of the Coq sources.

1.9 Notations

This section introduces some common mathematical notions and notations that will be used throughout this thesis.

Definition 1.9.1. *We let* $\mathbb{N}$ *denote the type of* natural numbers *(including* 0*), let* $\mathbb{Z}$ *denote the type of* integers*, and let* $\mathbb{Q}$ *denote the type of* rational numbers*. We let* $i \mid j$ *denote that* $i \in \mathbb{N}$ *is a* divisor *of* $j \in \mathbb{N}$*.*

Definition 1.9.2. *We let* Prop *denote the type of* propositions*, and let* bool *denote the type of* Booleans *whose elements are* true *and* false*. Most propositions we consider have a corresponding Boolean-valued decision function. In Coq we use type classes to keep track of these correspondences, but in this thesis we leave these correspondences implicit.*

Definition 1.9.3. *We let* $\mathsf{option}\ A$ *denote the* option type over A*, whose elements are inductively defined as either* $\bot$ *or* x *for some* $x \in A$*. Elements of the option type are denoted as* $x^? \in \mathsf{option}\ A$*. We implicitly lift operations to operate on the option type, and often omit cases of definitions that yield* $\bot$*. This is formally described using the* option monad *in the Coq formalization.*

Definition 1.9.4. *A* partial function f from A to B *is a function* $f : A \to \mathsf{option}\ B$*.*

Definition 1.9.5. *A partial function* f *is called* a finite partial function *or a* finite map *if its* domain $\mathsf{dom}\ f := \{x \mid \exists y \in B \,.\, f\,x = y\}$ *is finite. The type of finite partial functions is denoted as* $A \to_{\mathsf{fin}} B$*. The operation* $f[x := y]$ *yields* f *with the value* y *for argument* x*.*

Definition 1.9.6. *We let* 1 *denote the* unit type *whose only element is* $()$*.*

Definition 1.9.7. *We let* $A \times B$ *denote* the product of types A and B*. Given a* pair $(x, y) \in A \times B$*, we let* $(x, y)_{\mathbf{1}} := x$ *and* $(x, y)_{\mathbf{2}} := y$ *denote the* first *and* second projection *of* (x, y)*.*

Definition 1.9.8. *We let* $A + B$ *denote* the sum of types A and B*, whose elements are inductively defined as* $x_{\mathbf{l}}$ *for* $x \in A$ *or* $y_{\mathbf{r}}$ *for* $y \in B$*. We omit the subscripts* $_{\mathbf{l}}$ *and* $_{\mathbf{r}}$ *if they are clear from the context.*

Definition 1.9.9. *We let* $\mathsf{list}\ A$ *denote the* list type over A*, whose elements are inductively defined as either* ε *or* $x\,\vec{x}$ *for some* $x \in A$ *and* $\vec{x} \in \mathsf{list}\ A$*. We let* $x_i \in A$ *denote the* i*th element of a list* $\vec{x} \in \mathsf{list}\ A$ *(we count from 0). Lists are sometimes denoted as* $[\,x_0, \ldots, x_{n-1}\,] \in \mathsf{list}\ A$ *for* $x_0, \ldots, x_{n-1} \in A$*.*

We use the following operations on lists:

- *We often implicitly lift a function* $f : A_0 \to \cdots \to A_n$ *point-wise to the function* $f : \mathsf{list}\ A_0 \to \cdots \to \mathsf{list}\ A_n$*. The resulting list is truncated to the length of the smallest input list in case* $n > 1$*.*
- *We often implicitly lift a predicate* $P : A_0 \to A_{n-1} \to \mathsf{Prop}$ *to the predicate* $P : \mathsf{list}\ A_0 \to \cdots \to \mathsf{list}\ A_{n-1} \to \mathsf{Prop}$ *that guarantees that* P *holds for all (pairs of) elements of the list(s). The lifted predicate requires all lists to have the same length in case* $n > 1$*.*
- *We let* $|\vec{x}| \in \mathbb{N}$ *denote the length of* $\vec{x} \in \mathsf{list}\ A$*.*
- *We let* $\vec{x}_{[i,\,j)} \in \mathsf{list}\ A$ *denote the sublist* $x_i \ldots x_{j-1}$ *of* $\vec{x} \in \mathsf{list}\ A$*.*
- *We let* $x^n \in \mathsf{list}\ A$ *denote the list consisting of* n *times* $x \in A$*.*
- *We let* $(\vec{x}y^\infty)_{[i,\,j)} \in \mathsf{list}\ A$ *denote the sublist* $x_i \ldots x_{j-1}$ *of* $\vec{x} \in \mathsf{list}\ A$ *which is padded with* $y \in A$ *in case* $\vec{x}$ *is too short.*
- *Given lists* $\vec{x} \in \mathsf{list}\ A$ *and* $\vec{y} \in \mathsf{list}\ B$ *with* $|\vec{x}| = |\vec{y}|$*, we let* $\overrightarrow{xy} \in \mathsf{list}\ (A \times B)$ *denote the point-wise pairing of* $\vec{x}$ *and* $\vec{y}$*.*

Chapter 2

Subtleties of C

The semantics of C is oriented towards being efficiently implementable rather than being abstract in a mathematical sense. This is best captured by the following design choices from the standard's rationale [ISO03]:

- Trust the programmer.
- Do not prevent the programmer from doing what needs to be done.
- Make it fast, even if it is not guaranteed to be portable.

These design choices are both a blessing and a curse. They are the reason that C runs on almost every computer in existence, that C provides a high level of control, and that C code can be compiled to very efficient machine code.

The downside is a heavy burden on programmers. It is very easy for C programs to have bugs that make the program crash or behave badly in other ways. Furthermore, C programs can easily be developed with a too specific interpretation of the language in mind, giving portability and maintenance problems later.

The C standard achieves these goals by underspecification of the language. Instead of assigning a unique behavior to each program, it assigns a set of possible behaviors to each program. In the case of semantically illegal programs (those that have *undefined behavior* in C terminology) this set contains *all* possible behaviors (including letting the program crash). Surprisingly, correctly characterizing the set of programs that have undefined behavior is the hardest part of formalizing C.

This chapter describes the notions of underspecification used by the C11 standard, their consequences, as well as our formal treatment of them. This chapter moreover gives a brief introduction to C and illustrates a number of subtle forms of underspecification by means of example programs, their bizarre behaviors exhibited when compiled with widely used C compilers, and their treatment in $\mathrm{CH_2O}$.

2.1 Underspecification in the C standard

The C11 standard uses the following notions of underspecification [ISO12, 3.4]:

- **Implementation-defined behavior.** Program constructs for which the C11 standard delegates specification of their behavior to the compiler writer. For example, a compiler may use integers whose sizes are optimal for the target computing architecture. Implementation-defined behavior is consistent among each use of the program construct and should be documented by the compiler writer. The C11 standard only imposes some lower bounds.

- **Unspecified behavior.** Program constructs for which the C11 standard allows multiple behaviors. For example, each expression may be evaluated in the order deemed most efficient. A compiler writer does not have to document his choice, and his choice may vary for each use of the program construct. For example, evaluation of the same expression may be left to right in one situation, and right to left in another.
- **Undefined behavior.** Program constructs that are semantically illegal and for which the C11 standard imposes no requirements at all. Examples of undefined behavior are dereferencing a `NULL` or dangling pointer, signed integer overflow, and sequence point violations (modifying a memory location more than once in between two sequence points). In case of undefined behavior, the program may behave arbitrarily and is even allowed to crash.

Underspecification is used heavily by the C11 standard: 135 forms of implementation-defined behavior (including *locale specific behavior*), 58 forms of unspecified behavior, and 203 forms of undefined behavior are listed in [ISO12, Annex J].

The extensive use of underspecification in C shifts responsibilities from the compiler writer to the programmer:

- **Underspecification gives more freedom to compiler writers.** It allows for more effective optimizations, high run-time efficiency, and makes C easily portable among different computing architectures.
- **Underspecification is a burden for programmers.** Programmers have to ensure that their programs behave correctly under any choice of unspecified behavior and do not exhibit undefined behavior. Also, their assumptions about implementation-defined behavior should match up with the compiler.

Undefined behavior is often misunderstood. Let us consider dereferencing a `NULL` pointer as a running example. In more modern programming languages such as Java dereferencing a `NULL` pointer has a definite meaning: the semantics specifies that a `NullPointerException` is raised. Such an exception can be caught by error handling code, and if not, the program terminates with an error message.

In C the situation is different, dereferencing a `NULL` pointer has undefined behavior, which means that the standard imposes no requirements on the semantics whatsoever. A program in which a `NULL` pointer is dereferenced can therefore exhibit literally any behavior. For example, it may crash or it may yield unexpected results.

The main advantage of undefined behavior is higher run-time efficiency. Because the standard imposes no requirements on undefined behavior, a compiler can assume that a given program is free of undefined behavior. As a result, to compile a pointer dereference, a compiler does not have to generate code that checks whether the given pointer is unequal to `NULL` and raises an exception in case it is not. Instead, it is the programmer's responsibility to ensure that no `NULL` pointers are being dereferenced. The compiler will trust the programmer entirely on that.

It is important that programmers ensure that their programs do not exhibit any form of undefined behavior since undefined behavior may cause (security) bugs. Wang *et al.* [WCC$^+$12] and Dietz *et al.* [DLRA12] give an extensive overview. We consider an example by Corbet from the Linux kernel [Cor09]:

```
static unsigned int tun_chr_poll(struct file *file, poll_table *wait) {
  struct tun_file *tfile = file->private_data;
  struct tun_struct *tun = __tun_get(tfile);
  struct sock *sk = tun->sk;
  unsigned int mask = 0;
  if (!tun) return POLLERR;
  ...
```

In the first statement marked red, the `->` operator is used to dereference the struct pointer `tun` and access its field `sk` (`tun->sk` stands for `(*tun).sk`) Only afterwards, in the second statement marked red, it is checked that `tun` is not `NULL`.

What has been forgotten here is that dereferencing a `NULL` pointer has undefined behavior. Since a compiler may rightfully assume programs to be free of undefined behavior, it is allowed to assume that `tun` is not `NULL` and optimize out the 'superfluous' `NULL` check (GCC actually does do so). Without the `NULL` check, an attacker can run the rest of the function with `tun` pointing to the 0 address (which is a valid address in the Linux kernel), leading to privilege escalation [Cor09, WCC$^+$12].

Note that even if the current version of a compiler does not optimize out the `NULL` check, future versions of the same compiler may do so without notice.

2.2 Treating underspecification formally

Modeling underspecification formally is folklore:

- **Implementation-defined behavior is modeled by parameterization.** The entire CH$_2$O semantics is parameterized by a record that describes properties such as sizes and endianness of integers.
- **Unspecified behavior is modeled by non-determinism.** We use a non-deterministic small-step operational semantics. Selection of possible redexes is described using evaluation contexts [FFKD87].
- **Undefined behavior is modeled by the absence of a semantics.** We use a reduction to a special $\overline{\mathsf{undef}}$ state in the small-step operational semantics. This $\overline{\mathsf{undef}}$ state should be interpreted as "any behavior is allowed".

In order to prove that a program satisfies a specification, one has to prove that the specification is satisfied for all reduction paths through the small-step operational semantics, and that none of these reduction paths result into a $\overline{\mathsf{undef}}$ state (that is, one has to prove that the program is free of undefined behavior).

Because implementation-defined behavior is modeled by parameterization, one can either verify a given program with respect to a specific compiler (by instantiating the semantics with an implementation environment corresponding to that particular compiler), or verify a given program with respect to all compilers (by quantifying over all implementation environments). All metatheoretical results in this thesis are established for all CH_2O implementation environments. However, to reason about concrete programs, one often has to take certain implementation-defined properties into account, such as lower bounds on integer sizes.

2.3 Compiler independent semantics

There is an important difference between a semantics of the C standard and a semantics of a specific C compiler. In this section we explain this difference in the context of the CompCert compiler by Leroy [Ler09b]. CompCert compiles C code into assembly code (PowerPC, ARM and x86) and is written and verified in Coq.

The compiler correctness statement of CompCert involves a semantics of C and a semantics of the assembly language. The correctness statement is roughly:

> If the generated assembly code has a certain behavior (with respect to the assembly semantics), then the source C code also has that behavior (with respect to the C semantics).

This statement seems to be in the wrong direction, but it is not. The source C program may have multiple behaviors because of non-determinism (for example, due to different evaluation orders of side-effects in expressions), in which case it makes no sense to require the assembly code to exhibit all of these behaviors. Instead, each behavior the assembly exhibits should correspond to a behavior of the source C program. If the source C program has undefined behavior (*i.e.* it has all behaviors), the assembly code is allowed to exhibit any behavior (including crashing). The technical term for the above correctness statement is a *backward simulation* [Ler09a].

One could use the CompCert semantics to verify properties of actual C programs. In case one uses the CompCert compiler to compile the given program, one can be sure that the proven properties will hold for the generated assembly code too. However, verification against the CompCert semantics does not give reliable guarantees when the program is compiled using a different compiler such as GCC or Clang.

Like any compiler, CompCert has to make choices for implementation-defined behavior (for example integer representations). Furthermore, due to its intended use for embedded systems, CompCert gives a semantics to some undefined behaviors of C11 (such as violations of effective types) and compiles those in a faithful manner. When verifying properties of a program with respect to the CompCert semantics one can therefore make explicit use of behaviors that are defined by the CompCert semantics but undefined by C11. Note that although CompCert gives defined behavior to some undefined constructs, it still assigns undefined behavior to many illegal constructs such as dereferencing a `NULL` pointer and division by zero.

On the contrary, CH_2O intends to be a formal version of the C11 standard. We therefore have to ensure that if one proves a property of a program with respect to the CH_2O semantics, that property will hold when the program is compiled with *any* C11 compliant compiler. We thus have to take *all* unspecified and undefined behavior into account, even if that makes the semantics more complex, whereas CompCert may (and even has to) make specific choices for unspecified and undefined behavior. We have to

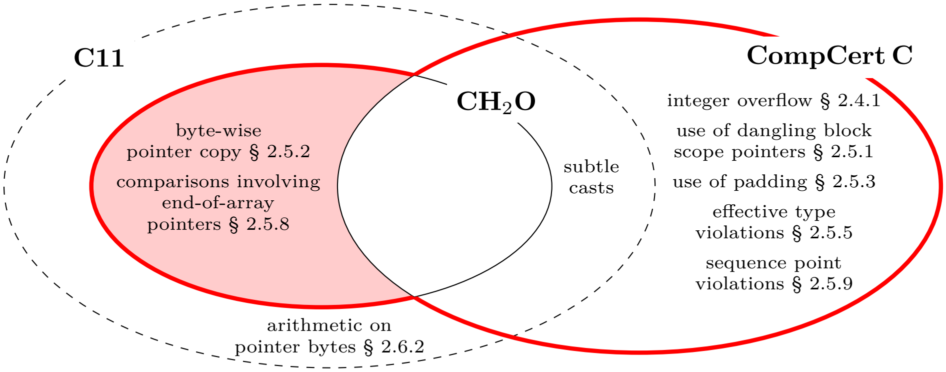

C11
CompCert C
CH$_2$O
byte-wise pointer copy § 2.5.2
comparisons involving end-of-array pointers § 2.5.8
subtle casts
integer overflow § 2.4.1
use of dangling block scope pointers § 2.5.1
use of padding § 2.5.3
effective type violations § 2.5.5
sequence point violations § 2.5.9
arithmetic on pointer bytes § 2.6.2

Figure 2.1: In the paper [KLW14], we have extended CompCert C with the behaviors in the shaded area to make it an instance of CH$_2$O. Each set in this diagram contains the programs that according to the semantics have defined behavior. Since the C11 standard is subject to interpretation, we draw it with a dashed line.

do so because the actual behavior of undefined and unspecified constructs differs from compiler to compiler, different versions of the same compiler, and different hardware, without that difference being documented at all.

Although CH$_2$O should describe all unspecified and undefined behaviors, we have more leeway for implementation-defined behavior. Since implementation-defined behavior is documented for each compiler, one can compare the compiler documentation with CH$_2$O. The goal of CH$_2$O is nonetheless to describe as many implementations as possible, but for example, implementations with legacy integer representations (such as sign-magnitude or 1's complement) are not relevant for modern practice.

For widely used compilers such as GCC and Clang, we are of course unable to give any *formal* guarantees that a correctness proof against the CH$_2$O semantics ensures correctness of the generated assembly code. After all, these compilers do not have a formal semantics. We can only argue that the CH$_2$O semantics has at least as much undefined behavior as the C11 standard, and assuming these compilers "implement the C11 standard" accordingly, correctness morally follows.

Since parts of the C11 standard are incomplete or ambiguous it is not always clear whether a given program has undefined behavior or not. The message [Mac01] on the standard committee's mailing list, Defect Reports #236, #260 and #451 [ISO], and the example programs in this chapter indicate various unclear parts. Since we do not know how compiler writers interpret such unclear parts of the standard, CH$_2$O errs on the side of caution: it makes certain behaviors undefined that some people deem defined according to the standard.

As a formal means of validation we intend to prove that the CompCert C semantics is an instance of CH$_2$O in future work (see also Section 11.2.2). That means:

- If a program has a certain defined behavior in CompCert C, the program has that behavior in CH$_2$O too. The program may have more defined behaviors (or even undefined behavior, which subsumes all behaviors) in CH$_2$O.

- If a program has undefined behavior in CompCert C, the program has undefined behavior in CH_2O too.

In earlier versions of CompCert this used to be not the case. Comparisons involving end-of-array pointers and performing a byte-wise copy of a pointer used to have undefined behavior in the CompCert semantics whereas both had defined behavior in CH_2O. In the paper [KLW14], we have therefore modified the CompCert semantics and compiler proofs accordingly. Since these modifications are inspired by CH_2O, we will not explicitly treat these in the context of CompCert in this thesis. Section 6.3.1 describes our formal treatment of end-of-array pointers and Section 5.5 describes our formal treatment of byte-wise copying of pointers in CH_2O.

Figure 2.1 gives an overview of the differences between CompCert, CH_2O and the C11 standard. It is important to keep in mind that this figure is not about features, but about *behaviors* of features that are both in CompCert and CH_2O.

2.4 Introduction to types in C

This section gives a brief introduction to the various data types in C. Those already familiar with C can easily skip this section.

2.4.1 Integer types

The C language provides the following integer types with increasing sizes:

`char` **`short`** **`int`** **`long`** **`long long`**

The type **`char`** constitutes a single byte and is able to hold one character of the local character set. All other integer types have implementation-defined sizes that are subject to some lower bounds [ISO12, 5.2.4.2.1]. The type **`int`** is intended to have the natural size for integer arithmetic on the target computing architecture.

All of these types have signed and unsigned variants which can be obtained using the **`signed`** and **`unsigned`** qualifiers. Signed integers include the negative cone whereas unsigned integers are non-negative. Unsigned integers obey the laws of arithmetic modulo 2^n where n is the number of bits. If the **`signed`** or **`unsigned`** qualifier is not given, the type is considered signed. An exception is **`char`** whose signedness is implementation-defined [ISO12, 6.2.5p15].

The exact rules for integer operators such as addition, multiplication and shifts are subtle. Most notably, overflow of arithmetic operations on signed integers has undefined behavior [ISO12, 6.5p5], which not all C programmers are aware of. Consider the following function that takes an **`int`** and returns an **`int`**:

```
int f(int x) { return x < x + 1; }
```

If the function `f` is called with the maximal value `INT_MAX` of the integer type **`int`** the addition `x + 1` will overflow.

As a result of undefined behavior of signed integer overflow, a compiler is allowed to optimize the function `f` to always return `1`. Indeed, when compiled with `gcc -O2` (version 4.9.2), the following program prints `2147483647 < -2147483648 = 1`.

```
printf("%d < %d = %d\n", INT_MAX, INT_MAX + 1, f(INT_MAX));
```

The previous example may seem artificial, but widely used compilers such as gcc use undefined behavior of integer overflow to optimize loops [Tay08].

In case an operator is applied to operands with different types, both operands are implicitly converted to a common type. These conversions are known as the *integer promotions* [ISO12, 6.3.1.1p2] and *usual arithmetic conversions* [ISO12, 6.3.1.8p1].

In order to avoid information loss, the rule of thumb is that the operand with the smallest type is converted into the type of the other operand. The situation is quite complicated if both signed and unsigned operands are involved. We further discuss these conversions in Section 3.2.

2.4.2 Pointer types

A pointer is a value that represents the address of an object in memory. In the example below p is declared to be a pointer to an integer.

```
int x = 10, y = 10;
int *p;
p = &x;               // p points to the variable x
printf("%d\n", *p);   // prints the value of x, i.e. 10
*p = 14;              // the variable x becomes 14
p = &y;               // p points to the variable y
```

The *address of operator* & yields a pointer to its operand. It can only be applied to objects that reside in memory (called *l-values* in C terminology) and not to constants, results of function calls and arithmetic operators, *etc.* The *dereferencing operator* * accesses the value its operand points to.

The declaration **int** *p declares p to have type **int***. In the declaration, the symbol * is grouped together with the variable name p instead of being grouped with the type **int**. This notation makes it possible to shorten declarations. For example, the first three lines of the example can be combined as follows:

```
int x = 10, y = 10, *p = &x;
```

The NULL pointer is a distinct pointer that does not point to any object. It is often used in a way similar to the ⊥ element of the option type. In C both the overloaded constant 0 and the macro NULL [ISO12, 7.19p3] denote the NULL pointer. Since NULL does not point to any object, dereferencing it has undefined behavior.

A *indeterminate* pointer is a pointer that has not been initialized or that points to an object that has reached the end of its lifetime (a *dangling* pointer). For example, the following function returns a dangling integer pointer.

```
int *f() { int x; return &x; }
```

Here, the lifetime of the object x has ended after the function f returns, and in turn &x becomes indeterminate.

Indeterminate pointers are essentially different from NULL pointers. Whereas it is only undefined behavior to dereference a NULL pointer, any operation on indeterminate pointers, including pointer comparison, has undefined behavior.

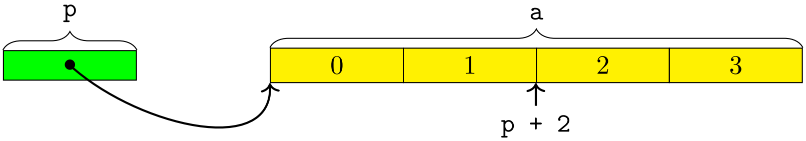
2.4.3 Array types

Array and pointer types are tightly connected in C. Let us demonstrate this by means of an example.

```
int a[4] = { 0, 1, 2, 3 };
printf("%d\n", a[2]);        // prints the value 2
int *p = a;                  // p points to the first element of a
printf("%d\n", *(p + 2));    // prints the value 2
printf("%d\n", p[3]);        // prints the value 3
```

The above code declares an array of 4 integers named `a`. The indexing operator `[`*i*`]` is used to refer to the *i*th element of the array.

Expressions of array type act like like expressions of pointer type that refer to the first element of the array. In the example `p` thus points to `a[0]`:



Pointer arithmetic is used to *move* a pointer through an array. In the example, the pointer `p + 2` points to `a[2]`. For that reason, the C11 standard as well as CH_2O define array indexing `e1[e2]` as `*(e1 + e2)`[1]. Pointers can thus be treated as if they were arrays, for example, `*(p + 3)` can also be written as `p[3]`. More surprisingly, any object can be treated as an array of size 1 [ISO12, 6.5.6p7].

Contrary to arrays in languages like Java, arrays in C do not contain their size. Accessing an array outside its bounds yields undefined behavior. This means that programmers often have to use a separate function argument for the array size. Alternatively, one can use a terminator. In particular, strings are represented as **char** arrays that are terminated with the *null character* `\0`. The length can then be computed by looking up the first occurrence of `\0`.

2.4.4 Void type and void pointers

The **void** type of C is used in two entirely unrelated ways. First of all, it is used for functions without a return value. For example, the function `swap` below takes two pointers and swaps the values they point to. This function does not return a value and is thus only called for its side-effects.

[1] Due to commutativity of addition, `e1[e2]` can be written as `e2[e1]`. This is often used to obfuscate code.

```
void swap (int *p, int *q) {
  int x = *q; // this temporary variable holds the old value of *q
  *q = *p;    // q points to the old value of *p
  *p = x;     // p points to the old value of *q
}
```

Secondly, the type `void*` is used for pointers to objects of unspecified type. Pointers of any type can be cast to a `void*` pointer and back. In this context, `void*` thus acts as a universal pointer type.

2.4.5 Struct types

Struct types are similar to product types in mathematics or record types in many other programming languages. For example:

```
struct rational { int numerator; int denominator; };
struct rational one = { .numerator = 1, .denominator = 1 };
```

The `struct` keyword introduces a new struct type named `rational` representing rational numbers. We call `numerator` and `denominator` the *fields* of the struct. The second line creates a variable named `one` of type `struct rational`.

The operator `.` is used to access a field of a struct. For example, consider the function `negate` that yields the additive inverse of a rational number.

```
struct rational negate(struct rational q) {
  q.numerator = -q.numerator;
  return q;
}
```

Passing and returning of struct values to functions is *call-by-value* instead of *call-by-reference*. This means that passing a struct to a function results in the actual value of the argument being copied into the callee's function variable. Changes made to the function variable have no effect on the argument. For example, after the call `negate(one)`, the variable `one` remains unchanged.

In case call-by-reference passing is needed, pointers to structs can be used. Pointers to structs are essential in recursive data structures, for example:

```
struct list { int head; struct list *tail; };
```

The above code declares a type `struct list` representing linked lists of integers. The field `tail` is a pointer to the tail instead of the tail itself. The function `length` computes the length of a linked list.

```
int length(struct list *p) {
  if(p) {
    return 1 + length(p->tail);
  } else return 0;
}
```

Since field indexing of pointers to structs is commonplace, C provides the notation `p->fieldname` as a mnemonic for `(*p).fieldname`.

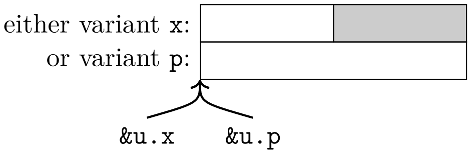
2.4.6 Union types

Union types are related to disjoint unions in mathematics or sum types in many other programming languages. The syntax for declaring a new union type is similar to the syntax for declaring struct types:

```
union short_or_pointer { short x; int *p; };
```

The `union` keyword introduces a new union type named `short_or_pointer` whose values is either a short or a pointer to an integer. In the example, we call `x` and `p` the *variants* of the union, which can be accessed using the `.` operator.

Although unions are related to sum types, unions are geared towards implementation and therefore quite different. A union occupies a single memory area that holds just the value of the current variant. It is the programmer's job to keep track of which variant is in use. It is therefore often said that unions are *untagged* instead of *tagged*. The representation of an object `u` of type `union short_or_pointer` may look like:



Since the size of one variant may be shorter than the other, there may be unused storage (called *padding*). In case a union `short_or_pointer` contains a value of the variant `q`, the part marked gray is unused.

A particular surprising aspect of unions is that pointers to both variants can be used in expressions. For example:

```
union short_or_pointer u;
u.x = 10;          // u has variant x with value 10
short *q = &u.x;   // q points to the x variant of u -> OK
int **qq = &u.p;   // qq points to the p variant of u -> OK
*qq = 0;           // u is accessed via a pointer to variant p -> bad
```

The pointers `q` and `qq` point to the same address in memory. Both the pointers `q` and `qq` are valid, but the rule of thumb is that only pointers to the current variant of a union may be used (the pointer `q` to the variant `x` in the example). However, as we will show in Section 2.5.5 and 2.5.6 the exact rules of C11 are more complicated.

2.5 Underspecification covered by CH_2O

This section illustrates a number of subtle forms of underspecification in C by means of example programs, their bizarre behaviors exhibited by widely used C compilers, and their treatment in CH_2O. Many of these examples involve delicacies due to the interaction between the following two ways of accessing data:

- In a *high-level* way using arrays, structs and unions.
- In a *low-level* way using unstructured and untyped byte representations.

The main problem is that compilers use a high-level view of data access to perform optimizations whereas both programmers and traditional memory models expect data access to behave in a concrete low-level way.

2.5.1 Non-local control and block scope variables

The C language allows unrestricted **goto**s which (unlike **break** and **continue**) may not only jump out of block scopes, but can also jump into block scopes (even **if** statements and loops). Block scopes may contain local variables, also called *block scope (local) variables*, which can be "taken out of their block" by keeping a pointer to them. The delicate part is that when leaving a block scope, the lifetime of its local variables expires. As a result, pointers to these local variables become indeterminate. Consider the following example where this is the case:

```
int *p = NULL;
l: if (p) {
  return (*p);
} else {
  int j = 10;
  p = &j;
  goto l;
}
```

When the label `l` is encountered initially, the variable `p` is `NULL`, and execution continues in the block where `p` is assigned to point to `j`. After execution of the **goto** `l`, the block containing `j` is left, and the lifetime of `j` expires. In turn, the conditional **if**`(p)` on a dangling pointer has undefined behavior.

This example indicates that a **goto** could have side-effects on the formal memory. These side-effects include allocation of local variables that have entered the scope, as well as deallocations of local variables that have gone out of scope. Although a compiler can ignore these allocation issues (it is the programmer's job to ensure that no dangling pointers are being used), a semantics cannot. Notice that also **return**, **break** and **continue** statements may create dangling pointers.

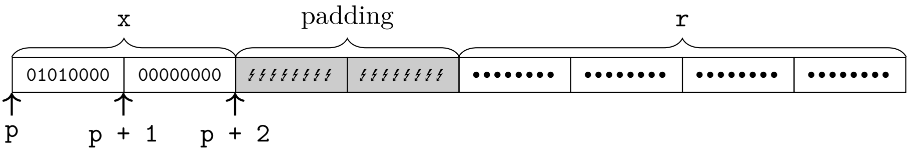
To accurately describe these side-effects in our operational semantics, we use a data structure based on Huet's *zipper* [Hue97] to store the location of the substatement that is being executed, as well as the program stack. In order to allow pointers to local variables, the stack contains references to the value of each variable instead of the value of each variable itself. Execution of all forms of control, including non-local control, is modeled by traversal through the zipper. Note that the goal of this traversal is not so much to *search* for the label, but much more to incrementally *calculate* the required allocations and deallocations.

2.5.2 Byte-level operations and object representations

Apart from *high-level* access to objects in memory by means of typed expressions, C also allows *low-level* access by means of byte-wise manipulation. Each object of type τ can be interpreted as an **unsigned char** array of length `sizeof`(τ), which is called the *object representation* [ISO12, 6.2.6.1p4]. Let us consider:

```
struct S { short x; short *r; } s1 = { 10, &s1.x };
unsigned char *p = (unsigned char*)&s1;
```

On 32-bit computing architectures such as x86 (with `_Alignof(short*)`$=4$), the object representation of `s1` might be:



The above object representation contains a hole due to *alignment* of objects. The bytes belonging to such holes are called *padding bytes.*

Alignment is the way objects are arranged in memory. In modern computing architectures, accesses to addresses that are a multiple of a word sized chunk (often a multiple of 4 bytes on a 32-bit computing architecture) are significantly faster due to the way the processor interacts with the memory. For that reason, the C11 standard has put restrictions on the addresses at which objects may be allocated [ISO12, 6.2.8]. For each type τ, there is an implementation-defined integer constant `_Alignof`(τ), and objects of type τ are required to be allocated at addresses that are a multiple of that constant. In case `_Alignof(short*)`$=4$, there are thus two bytes of padding in between the fields of `struct S`.

An object can be copied by copying its object representation. For example, the struct `s1` can be copied to `s2` as follows:

```
struct S { short x; short *r; } s1 = { 10, &s1.x };
struct S s2;
for (size_t i = 0; i < sizeof(struct S); i++)
  ((unsigned char*)&s2)[i] = ((unsigned char*)&s1)[i];
```

In the above code, `size_t` is an unsigned integer type, which is able to hold the results of the **sizeof** operator [ISO12, 7.19p2].

Manipulation of object representations of structs also involves access to padding bytes, which are not part of the high-level representation. In particular, in the example the padding bytes are also being copied. The problematic part is that padding bytes have indeterminate values, whereas in general, reading an indeterminate value has undefined behavior (for example, reading from an uninitialized **int** variable is undefined). The C11 standard provides an exception for **unsigned char** [ISO12, 6.2.6.1p5], and the above example thus has defined behavior.

The precise treatment of indeterminate values by the C11 standard is ambiguous, see Section 2.6.1. To that end, our memory model is conservative and uses a symbolic representation of bits (see Definition 5.3.1 on page 71) to distinguish determinate and indeterminate memory. This way, we can precisely keep track of the situations in which access to indeterminate memory is permitted.

2.5.3 Padding of structs and unions

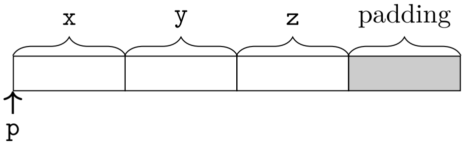
The following excerpt from the C11 standard points out another challenge with respect to padding bytes [ISO12, 6.2.6.1p6]:

> When a value is stored in an object of structure or union type, including in a member object, the bytes of the object representation that correspond to any padding bytes take unspecified values.

Let us illustrate this difficulty by an example:

```
struct S { char x; char y; char z; };
void f(struct S *p) { p->x = 0; p->y = 0; p->z = 0; }
```

On architectures with `sizeof(struct S) = 4`, objects of type `struct S` have one byte of padding. The object representation may be as follows:



Instead of compiling the function `f` to three store instructions for each field of the struct, the C11 standard allows a compiler to use a single instruction to store zeros to the entire struct. This will of course affect the padding byte. Consider:

```
struct S s = { 1, 1, 1 };
((unsigned char*)&s)[3] = 10;
f(&s);
return ((unsigned char*)&s)[3];
```

Now, the assignments to fields of `s` by the function `f` affect also the padding bytes of `s`, including the one `((unsigned char*)&s)[3]` that we have assigned to. As a consequence, the returned value is unspecified.

From a high-level perspective this behavior makes sense. Padding bytes are not part of the abstract value of a struct, so their actual value should not matter. However, from a low-level perspective it is peculiar. An assignment to a specific field of a struct affects the object representation of parts not assigned to.

None of the currently existing C formalizations describes this behavior correctly. In our tree based memory model we enforce that padding bytes always have an indeterminate value, and in turn we have the desired behavior implicitly. Note that if the function call `f(&s)` would have been removed, the behavior of the example program remains unchanged in CH_2O.

2.5.4 Aliasing of pointers

A drawback for efficient compilation of programming languages with pointers is *aliasing*. Aliasing describes a situation in which multiple pointers refer to the same object. In the following example the pointers `p` and `q` are said to be *aliased*.

```
int x; int *p = &x, *q = &x;
```

The problem of aliased pointers is that writes through one pointer may effect the result of reading through the other pointer. The presence of aliased pointers therefore often disallows one to change the order of instructions. For example, consider:

```
int f(int *p, int *q) {
  int z = *q; *p = 10; return z;
}
```

When `f` is called with pointers `p` and `q` that are aliased, the assignment to `*p` also affects `*q`. As a result, one cannot transform the function body of `f` into the shorter `*p = 10; return (*q);`. The shorter function will return 10 in case `p` and `q` are aliased, whereas the original `f` will always return the original value of `*q`.

Unlike this example, there are many situations in which pointers can be assumed *not* to alias. It is essential for an optimizing compiler to determine where aliasing cannot occur, and use this information to generate faster code. The technique of determining whether pointers can alias or not is called *alias analysis*.

2.5.5 Effective types

In *type-based alias analysis*, type information is used to determine whether pointers can alias or not. Consider the following example:

```
short g(int *p, short *q) {
  short z = *q; *p = 10; return z;
}
```

Here, a compiler is allowed to assume that `p` and `q` are not aliased because they point to objects of different types. The compiler is therefore allowed to transform the function body of `g` into the shorter `*p = 10; return (*q);`.

The peculiar thing is that the C type system does not statically enforce the property that pointers to objects of different types are not aliased. A union type can be used to create aliased pointers to different types:

```
union int_or_short { int x; short y; } u = { .y = 3 };
int *p = &u.x;    // p points to the x variant of u
short *q = &u.y;  // q points to the y variant of u
return g(p, q);   // g is called with aliased pointers p and q
```

The above program is valid according to the rules of the C11 type system, but has undefined behavior during execution of `g`. This is caused by the standard's notion of *effective types* [ISO12, 6.5p6-7] (also called *strict-aliasing restrictions*) that assigns undefined behavior to incorrect usage of aliased pointers to different types.

We will inline part of the function body of `g` to indicate the incorrect usage of aliased pointers during the execution of the example.

```
union int_or_short { int x; short y; } u = { .y = 3 };
int *p = &u.x;    // p points to the x variant of u
short *q = &u.y; // q points to the y variant of u
// g(p, q) is called, the body is inlined
short z = *q;     // u has variant y and is accessed through y -> OK
*p = 10;          // u has variant y and is accessed through x -> bad
```

The assignment `*p = 10` violates the rules for effective types. The memory area where `p` points to contains a union whose variant is `y` of type **`short`**, but is accessed through a pointer to variant `x` of type **`int`**. This causes undefined behavior.

Effective types form a clear tension between the low-level and high-level way of data access in C. The low-level representation of the memory is inherently untyped and unstructured and therefore does not contain any information about variants of unions. However, the standard treats the memory as if it were typed.

Most existing C formalizations (most notably Norrish [Nor99], Leroy *et al.* [Ler09b, LABS12] and Ellison and Roşu [ER12b]) use an unstructured untyped memory model where each object in the formal memory model consists of an array of bytes. These formalizations therefore cannot assign undefined behavior to violations of the rules for effective types. Our memory model is based on structured well-typed trees: effective types are modeled by the state of the trees in the memory model.

2.5.6 Type-punning

Despite the rules for effective types, it is under certain conditions nonetheless allowed to access a union through another variant than the current one. Accessing a union through another variant is called *type-punning*. For example:

```
union int_or_short { int x; short y; } u = { .x = 3 };
printf("%d\n", u.y);
```

This code will reinterpret the bit representation of the **`int`** value `3` of `u.x` as a value of type **`short`**. The reinterpreted value that is printed is implementation-defined (on architectures where **`short`**s do not have trap values).

Since the C11 standard is ambiguous about the exact conditions under which type-punning is allowed[2], we follow the interpretation by the GCC documentation [GCC]:

> Type-punning is allowed, provided the memory is accessed through the union type.

According to this interpretation the above program indeed has implementation defined behavior because the variant `y` is accessed via the expression `u.y` that involves the variable `u` of the corresponding union type.

[2] The term *type-punning* merely appears in a footnote [ISO12, footnote 95]. There is however the related *common initial sequence* rule [ISO12, 6.5.2.3], for which the C11 standard uses the notion of *visible*. This notion is not clearly defined either.

However, according to this interpretation, type-punning via a pointer to a specific variant of a union type yields undefined behavior. This is in agreement with the rules for effective types. For example, the following program has undefined behavior.

```
union int_or_short { int x; short y; } u = { .x = 3 };
short *p = &u.y;
printf("%d\n", *p);
```

We formalize the interpretation of C11 by GCC by decorating pointers and l-values to subobjects with annotations (Definition 5.2.3 on page 65). When a pointer to a variant of some union is stored in memory, or used as the argument of a function, the annotations are changed to ensure that type-punning no longer has defined behavior via that pointer. In Section 5.7 we prove that this approach is correct by showing that a compiler can perform type-based alias analysis (Theorem 5.7.2 on page 87).

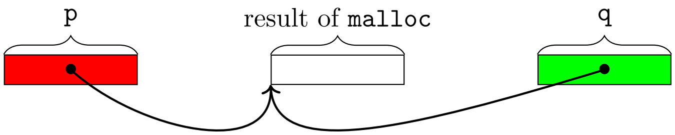
2.5.7 Indeterminate memory and pointers

A pointer value becomes indeterminate when the object it points to has reached the end of its lifetime [ISO12, 6.2.4] (it has gone out of scope, or has been deallocated). Dereferencing an indeterminate pointer has of course undefined behavior because it no longer points to an actual value. However, not many people are aware that using an indeterminate pointer in pointer arithmetic and pointer comparisons also yields undefined behavior. Consider:

```
int *p = malloc(sizeof(int)); assert (p != NULL);
free(p);
int *q = malloc(sizeof(int)); assert (q != NULL);
if (p == q) { // undefined, p is indeterminate due to the free
  *q = 10;
  *p = 14;
  printf("%d\n", *q); // p and q alias, expected to print 14
}
```

In this code `malloc(sizeof(int))` yields a pointer to a newly allocated memory area that may hold an integer, or yields a `NULL` pointer in case no memory is available. The function `free` deallocates memory allocated by `malloc`. In the example we assert that both calls to `malloc` succeed.

After execution of the second call to `malloc` it may happen that the memory area of the first call to `malloc` is reused: we have used `free` to deallocate it after all. This would lead to the following situation in memory:



Both GCC (version 4.9.2) or Clang (version 3.5.0) use the fact that `p` and `q` are obtained via different calls to `malloc` as a license to assume that `p` and `q` do not alias.

As a result, the value 10 of `*q` is inlined, and the program prints the value 10 instead of the naively expected value 14.

The situation becomes more subtle because when the object a pointer points to has been deallocated, not just the argument of `free` becomes indeterminate, but also all other copies of that pointer. This is therefore yet another example where high-level representations interact subtly with their low-level counterparts.

In our memory model we represent pointer values symbolically (Definition 5.2.3 on page 65), and keep track of memory areas that have been previously deallocated. The behavior of operations like `==` depends on the memory state, which allows us to accurately capture the described undefined behaviors.

2.5.8 End-of-array pointers

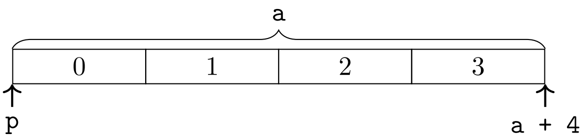
The way the C11 standard deals with pointer equality is subtle. Consider the following excerpt [ISO12, 6.5.9p6]:

> Two pointers compare equal if and only if [...] or one is a pointer to one past the end of one array object and the other is a pointer to the start of a different array object that happens to immediately follow the first array object in the address space.

End-of-array pointers are peculiar because they cannot be dereferenced, they do not point to any value after all. Nonetheless, end-of-array are commonly used when looping through arrays.

```
int a[4] = { 0, 1, 2, 3 };
int *p = a;
while (p < a + 4) { *p += 1; p += 1; }
```

The pointer `p` initially refers to the first element of the array `a`. The value `p` points to, as well as `p` itself, is being increased as long as `p` is before the end-of-array pointer `a + 4`. This code thus increases the values of the array `a`. The initial state of the memory is displayed below:



End-of-array pointers can also be used in a way where the result of a comparison is not well-defined. In the example below, the `printf` is executed only if `x` and `y` are allocated adjacently in the address space (typically the stack).

```
int x, y;
if (&x + 1 == &y) printf("x and y are allocated adjacently\n");
```

Based on the aforementioned excerpt of the C11 standard [ISO12, 6.5.9p6], one would naively say that the value of `&x + 1 == &y` is uniquely determined by the way `x` and `y` are allocated in the address space. However, the GCC implementers disagree[3]. They claim that Defect Report #260 [ISO] allows them to take the *derivation of a pointer value* into account.

In the example, the pointers `&x + 1` and `&y` are derived from unrelated objects (the local variables `x` and `y`). As a result, the GCC developers claim that `&x + 1` and `&y` may compare unequal albeit being allocated adjacently. Consider:

```
int compare(int *p, int *q) {
  // some code to confuse the optimizer
  return p == q;
}
int main() {
  int x, y;
  if (&x + 1 == &y) printf("x and y are adjacent\n");
  if (compare(&x + 1, &y)) printf("x and y are still adjacent\n");
}
```

When compiled with GCC 4.9.2, we have observed that only the string `x and y are still adjacent` is being printed. This means that the value of `&x + 1 == &y` is not consistent among different occurrences of the comparison.

Due to these discrepancies we assign undefined behavior to questionable uses of end-of-array pointers while assigning the correct defined behavior to pointer comparisons involving end-of-array pointers when looping through arrays (such as in the first example above). Our treatment is similar to our extension of CompCert [KLW14].

2.5.9 Sequence point violations and non-determinism

Instead of having to follow a specific execution order, the execution order of expressions is unspecified in C. This is a common cause of portability problems because a compiler may use an arbitrary execution order for each expression, and each time that expression is executed. Hence, to ensure correctness of a C program with respect to an arbitrary compiler, one has to verify that *each* possible execution order is free of undefined behavior and gives the correct result.

In order to make more effective optimizations possible (for example, delaying of side-effects and interleaving), the C standard does not allow an object to be modified more than once during the execution of an expression. If an object is modified more than once, the program has undefined behavior. We call this requirement the *sequence point restriction*. Note that this is not a static restriction, but a restriction on valid executions of the program. Let us consider an example:

```
int x, y = (x = 3) + (x = 4);
printf("%d %d\n", x, y);
```

[3] See `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=61502`

By considering all possible execution orders, one would naively expect this program to print `4 7` or `3 7`, depending on whether the assignment `x = 3` or `x = 4` is executed first. However, `x` is modified twice within the same expression, and thus both execution orders have undefined behavior. The program is thereby allowed to exhibit any behavior. Indeed, when compiled with `gcc -O2` (version 4.9.2), the compiled program prints `4 8`, which does not correspond to any of the execution orders.

The exact rules corresponding to the sequence point restriction are more lenient: an object may not be modified more than once or being read after being modified between two *sequence points* [ISO12, 6.5p2] instead of whole expressions. A sequence point occurs for example at the semicolon that terminates a full expression, before a function call, and after the first operand of the conditional `? :` operator [ISO12, Annex C]. The following expression thus has defined behavior:

```
int x, y = ((x = 3) ? (x = 4) : (x = 5));
```

In case assignments are hidden behind a function call, the sequence point restriction also does not apply. For example:

```
int assign(int *p, int y) { return *p = y; }
int main() {
  int x;
  assign(&x, 3) + assign(&x, 4);
  return x;
}
```

Due to the sequence point before the function calls to `assign`, no sequence point violation occurs. This program thus non-deterministically returns 3 or 4.

Our approach to handling non-determinism and sequence points is inspired by Norrish [Nor98] and Ellison and Roşu [ER12b]. Our small-step operational semantics uses evaluation contexts [FFKD87] to non-deterministically select a redex in a whole expression. Furthermore, each bit in memory carries a permission (Definition 4.3.2 on page 54) that is set to a special *locked* permission when a store has been performed. The memory model prohibits any access (read or store) to objects with locked permissions. At the next sequence point, the permissions of locked objects are changed back into their original permission, making future accesses possible again.

2.6 Other forms of underspecification

There are two ways in which the CH$_2$O semantics deviates from the C11 standard: some features are missing, and some behaviors are undefined in CH$_2$O that certain people consider defined according to the C11 standard. There are two main reasons why CH$_2$O makes more behaviors undefined:

- The standard is incomplete or ambiguous. In this case, compiler writers may have different interpretations of the standard text, and therefore may perform more aggressive optimizations than expected. So as to be compiler independent, CH$_2$O assumes the most restrictive reading by assigning undefined behavior.

- In order to avoid excessive non-determinism. In case underspecification leads to an excessive amount of non-determinism, the set of behaviors of even small programs may become exceedingly large. This may for example happen if one assigns a non-deterministic object representation to uninitialized variables.

  Too much non-determinism makes it impossible to naively implement an executable semantics that can be used to explore all behaviors of already small programs. Hence, we sometimes use symbolic representations and assign undefined behavior in some cases, which subsumes non-determinism.

There are also other technical reasons why excessive non-determinism is undesired. Namely, although the CH_2O semantics is non-deterministic, the memory operations are deterministic. Deterministic memory operations allow for convenient equational reasoning, as used extensively to reason about the separation algebra operations on memories. This would be more difficult with of non-deterministic operations.

This section describes some features of C that are either not supported by CH_2O, or that have more undefined behaviors in CH_2O than certain people might consider reasonable according to their reading of the C11 standard.

2.6.1 Integer representations of indeterminate memory

The type **`unsigned char`** has a special status in C, it is the type of bytes that constitute object representations (see Section 2.5.2). For that reason, in order to facilitate byte-wise copying of structs, the C11 standard allows one to access all bytes, even those that are indeterminate, using an expression of type **`unsigned char`**. However, it is unclear from the standard what is supposed to happen when we do not just store these bytes elsewhere, but for example print them[4]:

```
unsigned char i; // i is intentionally uninitialized
&i;              // i is not in a register (6.3.2.1p2)
printf("%d\n", i); printf("%d\n", i);
```

Does execution of this program exhibit undefined behavior? And if not, does it have to print the same number twice? Since the answers to these questions are unclear from the standard text, Defect Report #260 [ISO] considered a similar question:

> If an object holds an indeterminate value, can that value change other than by an explicit action of the program?

And the response of the standard committee was:

[4] The second line of the example circumvents another restriction of the C11 standard. Namely, Defect Report #338 [ISO] has noticed that on some architectures (*e.g.* IA-64) registers may hold trap values that do not exist in memory. Thus, for such architectures, programs cannot safely copy uninitialized variables of type **`unsigned char`** that reside in registers.

The C11 standard has repaired this problem by an ad-hoc workaround [ISO12, 6.3.2.1p2]: "if the lvalue designates an object of automatic storage duration that could have been declared with the register storage class, and that object is uninitialized, the behavior is undefined." We circumvent this workaround by taking the address `&i` of `i`.

> In the case of an indeterminate value all bit-patterns are valid representations and the actual bit-pattern may change without direct action of the program.

Although Defect Report #260 has been made obsolete by the C11 standard, the decision at that time was that the standard text did not need to be changed. The wording of all relevant parts of C99 and C11 remained identical.

We have attempted to resolve this unclarity by submitting a new defect report. This Defect Report is numbered #451 [ISO]. Apart from that, we have submitted a document [KW14] to the standard committee, and Freek Wiedijk has presented the problem during the 2014 meeting of the standard committee in Parma. The standard committee has reaffirmed its position on Defect Report #260 [ISO] in favor of more undefined behavior. However, our efforts have not led to a clarification of the standard text. To that end, CH$_2$O assigns undefined behavior to uses of indeterminate values of type `unsigned char` other than reads and stores (for example, adding 0 to an indeterminate value of type `unsigned char` is undefined in CH$_2$O).

It may seem academic to consider what happens when one uses uninitialized variables (one should initialize one's variables), but as explained in Section 2.5.2, indeterminate values appear naturally in the padding bytes of structs. In case indeterminate values are allowed to change arbitrarily, a program that temporarily dumps data from its memory (which may contains structs) into a file could not be implemented. For that reason, our Defect Report #451 [ISO] argues to give a definite meaning to these programs and clarify the standard text accordingly.

2.6.2 Integer representations of pointers

On actual computing architectures pointers are represented as integer values that refer to an offset into the address space. This fact is regularly exploited by code used in practice. For example integer representations of pointers are used for indexing into a hash table, and their use is common in the code of operating systems such as Linux.

The C11 standard describes (optional) integer types that can be used to cast a pointer to an integer. These integer types are called `intptr_t` and `uintptr_t` [ISO12, 7.20.1.4]. Not many properties about casting a pointer to such integers and *vice versa* are guaranteed. It is only guaranteed that if a valid `void` is cast to these types, the pointer obtained by casting back compares equal to the original pointer.

The CH$_2$O semantics uses an abstract memory model with symbolic pointer values and therefore fails to account for pointer to integer casts. Casting a pointer to an integer, and *vice versa*, has undefined behavior in the CH$_2$O semantics.

In CompCert [Ler09a], the situation is similar, but slightly relaxed. Integer types are not only allowed to hold numerical values, but also symbolic pointer values. Casting a pointer to an integer, and *vice versa*, preserves the symbolic structure of the pointer value in question. This approach is rather limited. Most arithmetic operations on these "symbolic pointers in integer disguise", like the arithmetic operations used to compute an index into a hash table, still have undefined behavior. Also, this approach invalidates some compiler optimizations [KHM$^+$15].

Kang *et al.* [KHM$^+$15] have proposed a memory model for C that uses both numerical and symbolic pointer representations. Initially each pointer is represented symbolically, but whenever the numerical representation of a pointer is needed (due to a pointer to integer cast), it is non-deterministically *realized*. Contrary to a memory model based solely on numerical pointer values, they show that their memory model allows desirable compiler optimizations. Compatibility of their approach with respect to the C11 standard and integration with struct and union types remains to be investigated. See also the discussion in Section 10.2.

Given that Defect Report #260 [ISO] allows a compiler to take the derivation of a pointer value into account, a natural question is whether that also holds for non-pointer values. Consider the following program (suggested by Marc Schoolderman):

```
int x = 30, y = 31;
int *p = &x + 1, *q = &y;
intptr_t i = (intptr_t)p, j = (intptr_t)q;
printf("%ld %ld %d\n", i, j, i == j);
```

When compiled with GCC 4.7.1, this may output:

```
140734814994316 140734814994316 0
```

Although the `i` and `j` have the same numeric value, they still compare as unequal. We have reported this problem to the GCC bug tracker[5]. In the reactions to our bug report, this behavior was unequivocally considered to be a bug, and has been changed in GCC 4.8. In other words, although some compiler writers think the license for optimizations that Defect Report #260 [ISO] gives them is justified, this example was too extreme for the GCC community to be taken in that light.

2.6.3 Finiteness of memory

The C programming language provides three ways to allocate memory. These correspond to the *storage durations* described in the C11 standard [ISO12, 6.2.4].

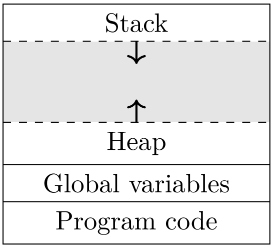
- **Static storage.** This storage duration is used for global and static local variables. Objects of this storage duration are allocated and initialized once prior to program startup, and their lifetime is the entire execution of the program.
- **Automatic storage.** This storage duration is used for function arguments and block scope local variables. Objects of this storage duration are allocated when a function is called, or an enclosed block scope is entered. The lifetime of these objects extends until the function returns, or the enclosed block scope is left (possibly by non-local control, see Section 2.5.1). In case a function is called recursively, new objects are allocated for each function invocation.
- **Allocated storage.** This storage duration is used for memory allocated via standard library functions like `malloc` [ISO12, 7.22.3.4] and `calloc` [ISO12, 7.22.3.2]. Objects of this storage duration should be deallocated explicitly using the standard library function `free` [ISO12, 7.22.3.3].

[5] See `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=54945`

All of these ways to allocate memory can fail if the machine has run out of memory. The C11 standard captures failure of obtaining memory of allocated storage duration by letting standard library functions like `malloc` or `calloc` return a `NULL` pointer. However, the standard does not account for failure to obtain memory of static or automatic storage duration in any way.

For the case of static storage, the standard's omission to address failure of obtaining memory does not have practical implications. Allocation is performed prior to program startup, and it is known statically how much memory will be allocated.

The C11 standard's omission to account for failure of obtaining memory of automatic storage is a serious issue. Most computing architectures organize their address space into two parts: the *heap* and the *stack*. A simplified version looks as follows:



The heap contains memory of allocated storage duration and the stack contains memory of automatic storage duration. When a function is called, the stack is extended with a frame containing the function arguments, the local variables, and the return address (the address of the instruction that has to be executed when the function is finished). The stack generally has a limited size, and too deep recursion results in so called *stack overflow*.

Memory protection on most general purpose computing architectures ensures an immediate system failure (in the form of a message like `segmentation fault`) in case of stack overflow. However, on micro controllers used in embedded computing this is often not the case. Stack overflow may thus overwrite parts of the heap, or even parts of the program code. An interesting real-life example is the Toyota unintended acceleration lawsuit [Sam14]. In this lawsuit it was claimed that stack overflow caused memory corruption, which resulted in vehicles to accelerate unexpectedly, and thereby even killed people.

The standard abstracts from implementation details such as the organization of the memory, and thereby allows implementations to organize their memory differently than the standard stack and heap organization. Although CH_2O abstracts from these details as well, we believe the standard should account for an abstract version of stack overflow. Unfortunately, stack overflow is not mentioned in any way by either the C11 standard [ISO12] or the standard's rationale [ISO03].

The CH_2O semantics thus uses a memory of unbounded size in which allocation of function arguments and block scope local variables cannot fail. The `malloc` function non-deterministically succeeds or yields `NULL`. Our treatment corresponds to the C11 standard, and is similar to Leroy's CompCert [Ler09b] (which even has an unbounded memory model on the level of assembly) and Ellison and Roşu [ER12b].

Verification of non-functional properties such as stack and memory consumption is an instance of a broader field of research that also considers time complexity and other resources a program may use. We will not treat verification of resource consumption in this thesis, but will finish this chapter with a small survey of the difficulties involved and the current state of the art in proof assistants for real-life languages.

When verifying assembly code, there is a close correspondence between stack and memory consumption in the formal semantics and memory consumption in the actual machine. Myreen [Myr08] takes bounds on stack usage into account when verifying assembly code. However, for C verification the situation is more challenging because a compiler is involved. The problem is that a compiler does not necessarily preserve stack consumption of the program it compiles. For example, inlining of functions and spilling of variables may increase the stack usage arbitrarily [Ler09a].

The CerCo compiler by Asperti *et al.* [AAA+11, AAB+13] addresses the problem by propagating information about stack usage and time complexity through all compilation phases of a simple verified C compiler to generate corresponding verification conditions in the C sources. Carbonneaux *et al.* use a similar approach in the context of the CompCert compiler [CHRS14] and provide a Hoare logic to establish bounds on stack usage and a verified stack usage analyzer om top of it. These solutions are however compiler specific and omit features of the C language.

Chapter 3

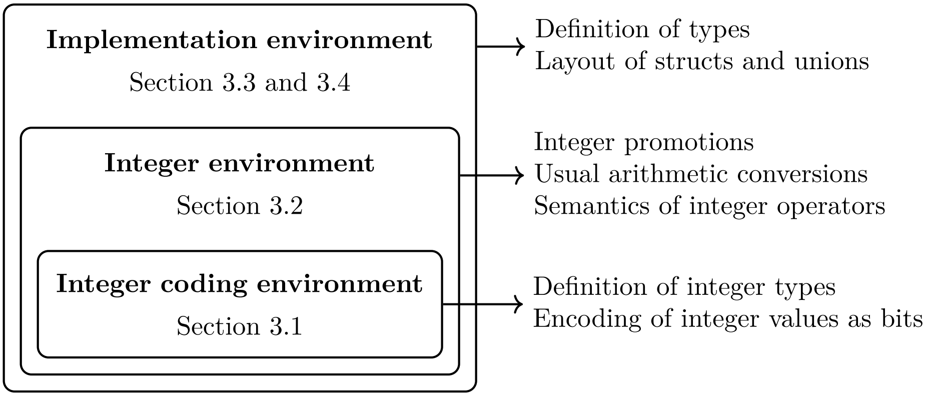
Types in C

Contrary to more modern typed programming languages such as Java, ML or Haskell, C has a rather weak type system that does not enjoy type safety. That means, well-typedness according to the C type system does not ensure the absence of run-time errors (such as out of bounds array accesses) nor the absence of run-time type errors related to effective types. Nonetheless, the C type system ensures the absence of basic mistakes such as using a wrongly named field of a struct, or using an integer where a pointer is expected (without an explicit cast).

This chapter describes the types of our language CH_2O core C. It supports integer, pointer, function pointer, array, struct, union and void types. More complicated types such as enum types and typedefs are defined in Chapter 7 by translation.

This chapter furthermore describes an abstract interface, called an *implementation environment*, that describes properties such as size and endianness of integers, and the layout of structs and unions. The entire CH_2O semantics will be parameterized by an implementation environment.

Implementation environments are defined modularly using a chain of nested environments. The structure of implementation environments, as well as the outline of this chapter, is depicted in the following diagram.


Implementation environment
Section 3.3 and 3.4
Definition of types
Layout of structs and unions
Integer environment
Section 3.2
Integer promotions
Usual arithmetic conversions
Semantics of integer operators
Integer coding environment
Section 3.1
Definition of integer types
Encoding of integer values as bits

3.1 Integer representations

This section describes the part of implementation environments corresponding to integer types and the encoding of integer values as bits. Integer types consist of a *rank* (char, short, int ...) and a *signedness* (signed or $\mathsf{unsigned}$). The set of available ranks as well as many of their properties are implementation-defined. We therefore abstract over the ranks in the definition of integer types.

Definition 3.1.1. Integer signedness *and* integer types *over ranks $k \in K$ are inductively defined as:*

$$\begin{aligned} si \in \mathsf{signedness} &::= \mathsf{signed} \mid \mathsf{unsigned} \\ \tau_\mathsf{i} \in \mathsf{inttype} &::= si\ k \end{aligned}$$

The projections are called $\mathsf{rank} : \mathsf{inttype} \to K$ *and* $\mathsf{sign} : \mathsf{inttype} \to \mathsf{signedness}$.

Definition 3.1.2. *An* integer coding environment with ranks K *consists of a total order* $(K, \subseteq)$ *of* integer ranks *having at least the following ranks:*

$$\mathsf{char} \subset \mathsf{short} \subset \mathsf{int} \subset \mathsf{long} \subset \mathsf{long\ long} \qquad \textit{and} \qquad \mathsf{ptr\_rank}.$$

It moreover has the following functions:

$$\begin{aligned} \mathsf{char\_bits} &: \mathbb{N}_{\geq 8} & \mathsf{endianize} &: K \to \mathsf{list\ bool} \to \mathsf{list\ bool} \\ \mathsf{char\_signedness} &: \mathsf{signedness} & \mathsf{deendianize} &: K \to \mathsf{list\ bool} \to \mathsf{list\ bool} \\ \mathsf{rank\_size} &: K \to \mathbb{N}_{>0} \end{aligned}$$

Here, $\mathsf{endianize}\ k$ *and* $\mathsf{deendianize}\ k$ *should be inverses,* $\mathsf{endianize}\ k$ *should be a permutation,* $\mathsf{rank\_size}$ *should be (non-strictly) monotone, and* $\mathsf{rank\_size\ char} = 1$.

Definition 3.1.3. *The judgment $x : \tau_\mathsf{i}$ describes that $x \in \mathbb{Z}$* has integer type τ_i.

$$\frac{-2^{\mathsf{char\_bits} * \mathsf{rank\_size}\ k - 1} \leq x < 2^{\mathsf{char\_bits} * \mathsf{rank\_size}\ k - 1}}{x : \mathsf{signed}\ k} \qquad \frac{0 \leq x < 2^{\mathsf{char\_bits} * \mathsf{rank\_size}\ k}}{x : \mathsf{unsigned}\ k}$$

The rank char is the rank of the smallest integer type, whose unsigned variant corresponds to bytes that constitute object representations (see Section 2.5.2). Its bit size is $\mathsf{char\_bits}$ (called `CHAR_BIT` in the standard library header files [ISO12, 5.2.4.2.1]), and its signedness $\mathsf{char\_signedness}$ is implementation-defined [ISO12, 6.2.5p15].

The rank $\mathsf{ptr\_rank}$ is the rank of the integer types `size_t` and `ptrdiff_t`, which are defined in the standard library header files [ISO12, 7.19p2]. The type `ptrdiff_t` is a signed integer type used to represent the result of subtracting two pointers, and the type `size_t` is an unsigned integer type used to represent sizes of types.

An integer coding environment can have an arbitrary number of integer ranks apart from the standard ones char, short, int, long, $\mathsf{long\ long}$, and $\mathsf{ptr\_rank}$. This way, additional integer types like those describe in [ISO12, 7.20] can easily be included.

The function $\mathsf{rank\_size}$ gives the byte size of an integer of a given rank. Since we require $\mathsf{rank\_size}$ to be monotone rather than strictly monotone, integer types with different ranks can have the same size [ISO12, 6.3.1.1p1]. For example, on many implementations **`int`** and **`long`** have the same size, but are in fact different.

The C11 standard allows implementations to use either sign-magnitude, 1's complement or 2's complement signed integers representations. It moreover allows integer representations to contain padding or parity bits [ISO12, 6.2.6.2]. However, since all current machine architectures use 2's complement representations, this is more of a historic artifact. Current machine architectures use 2's complement representations because these do not suffer from positive and negative zeros and thus enjoy unique representations of the same integer. Hence, CH_2O restricts itself to implementations that use 2's complement signed integers representations.

Integer representations in CH_2O can solely differ with respect to endianness (the order of the bits). The function $\mathsf{endianize}$ takes a list of bits in little endian order and permutes them accordingly. We allow $\mathsf{endianize}$ to yield an arbitrary permutation and thus we not just support big- and little-endian, but also mixed-endian variants.

Definition 3.1.4. *Given an integer type* τ_{i}, *the* integer encoding functions $\overline{\_ : \tau_{\mathsf{i}}} : \mathbb{Z} \to \mathsf{list\ bool}$ *and* $(\_)_{\tau_{\mathsf{i}}} : \mathsf{list\ bool} \to \mathbb{Z}$ *are defined as follows:*

$$\overline{x : si\ k} := \mathsf{endianize}\ k\ (x \text{ as little endian 2's complement})$$

$$(\vec{\beta})_{si\ k} := \text{of little endian 2's complement } (\mathsf{deendianize}\ k\ \vec{\beta})$$

Lemma 3.1.5. *The integer encoding functions are inverses. That means:*

1. *We have* $(\overline{x : \tau_{\mathsf{i}}})_{\tau_{\mathsf{i}}} = x$ *and* $|\overline{x : \tau_{\mathsf{i}}}| = \mathsf{rank\_size}\ \tau_{\mathsf{i}}$ *provided that* $x : \tau_{\mathsf{i}}$.
2. *We have* $\overline{(\vec{\beta})_{\tau_{\mathsf{i}}} : \tau_{\mathsf{i}}} = \vec{\beta}$ *and* $(\vec{\beta})_{\tau_{\mathsf{i}}} : \tau_{\mathsf{i}}$ *provided that* $|\vec{\beta}| = \mathsf{rank\_size}\ \tau_{\mathsf{i}}$.

3.2 Integer operators

We extend the interface of integer coding environments to describe the implementation-defined and undefined aspects of the semantics of integer operators.

Definition 3.2.1. C unary and binary operators *are inductively defined as:*

$$\begin{array}{rl@{\qquad}rl}
\circledcirc_c \in \mathsf{compop} ::= & \texttt{==} \mid \texttt{<=} \mid \texttt{<} & \circledcirc_s \in \mathsf{shiftop} ::= & \texttt{<<} \mid \texttt{>>} \\
\circledcirc_b \in \mathsf{bitop} ::= & \texttt{\&} \mid \texttt{|} \mid \texttt{\^{}} & \circledcirc \in \mathsf{binop} ::= & \circledcirc_c \mid \circledcirc_b \mid \circledcirc_a \mid \circledcirc_s \\
\circledcirc_a \in \mathsf{arithop} ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} & \circledcirc_u \in \mathsf{unop} ::= & \texttt{-} \mid \texttt{\~{}} \mid \texttt{!}
\end{array}$$

An integer implementation (Definition 3.2.5) describes the semantics of casts, and arithmetic $\circledcirc_a$ and shift $\circledcirc_s$ operators. The semantics of the comparison $\circledcirc_c$, bitwise $\circledcirc_b$, and unary $\circledcirc_u$ operators will be defined in terms of the other operations. The comparison `x != y` will be de-sugared into `!(x == y)`.

Implementations have significant leeway in how to implement integer operators. Most notably, signed overflow of arithmetic operations has undefined behavior [ISO12, 6.5p5], and if an integer is cast to a signed type that cannot represent it, either a signal is raised, or the resulting value is implementation-defined [ISO12, 6.3.1.3].

Likewise, the resulting value of a right shift of a negative integer is implementation-defined [ISO12, 6.5.7p4]. The C11 standard provides this amount of leeway because native machine instructions on different computing architectures tend to behave differently for corner cases.

Before being able to define the semantics of integer operators while taking these forms of underspecification into account, we have to address typing of operators. The typing rules of integer operators involve subtle implicit conversions. These conversions are performed in two steps:

1. The *integer promotions*: arguments of integer types whose rank is smaller than int are promoted to int [ISO12, 6.3.1.1p2].
2. The *usual arithmetic conversions*: the promoted arguments are converted to an integer of common type [ISO12, 6.3.1.8p1].

We define functions $\lceil \_ \rceil : \mathsf{inttype} \to \mathsf{inttype}$ and $\cup : \mathsf{inttype} \to \mathsf{inttype} \to \mathsf{inttype}$ that correspond to the two steps above. The type of an arithmetic operator $\circledcirc_a$ with arguments of τ_i and σ_i is $\lceil \tau_\mathsf{i} \rceil \cup \lceil \sigma_\mathsf{i} \rceil$.

The need for the integer promotions is motivated by the fact that many machines do not have fast/native instructions for integer operators on small integer types.

Definition 3.2.2. *The function* $\lceil \_ \rceil : \mathsf{inttype} \to \mathsf{inttype}$ *performs* integer promotions.

$$\lceil \tau_\mathsf{i} \rceil := \begin{cases} \mathsf{signed\ int} & \text{if } \mathsf{rank}\ \tau_\mathsf{i} \subseteq \mathsf{int} \text{ and } \mathsf{signed\ int} \text{ can represent all } \tau_\mathsf{i} \text{ values} \\ \mathsf{unsigned\ int} & \text{if } \mathsf{rank}\ \tau_\mathsf{i} \subseteq \mathsf{int} \text{ and } \mathsf{signed\ int} \text{ cannot represent all } \tau_\mathsf{i} \text{ values} \\ \tau_\mathsf{i} & \text{otherwise} \end{cases}$$

The C11 standard defines the *usual arithmetic conversions* as a binary operation on integer types. This definition is complicated and contains some implicit duplication due to symmetry, which is inconvenient when proving properties. To improve the situation, we show that the usual arithmetic conversion of integer types τ_i and σ_i is a least upper bound $\tau_\mathsf{i} \cup \sigma_\mathsf{i}$ in a join semi-lattice.

Definition 3.2.3. *The* usual arithmetic order $(\mathsf{inttype}, \subseteq)$ *is inductively defined as:*

1. *If* $k_1 \subseteq k_2$ *then* $si\ k_1 \subseteq si\ k_2$ *and* $\mathsf{signed}\ k_1 \subseteq \mathsf{unsigned}\ k_2$.
2. *If* $\mathsf{signed}\ k_2$ *can represent all* $\mathsf{unsigned}\ k_1$ *values, then* $\mathsf{unsigned}\ k_1 \subseteq \mathsf{signed}\ k_2$.

Lemma 3.2.4. *All integer types* τ_i *and* σ_i *have a least upper bound* $\tau_\mathsf{i} \cup \sigma_\mathsf{i}$ *in the usual arithmetic order* $(\mathsf{inttype}, \subseteq)$. *The upper bound* $\tau_\mathsf{i} \cup \sigma_\mathsf{i}$ *corresponds to the* usual arithmetic conversion *in the C11 standard:*

$$\begin{aligned} \mathsf{signed}\ k_1 \cup \mathsf{signed}\ k_2 &= \mathsf{signed}\ (k_1 \cup k_2) \\ \mathsf{unsigned}\ k_1 \cup \mathsf{unsigned}\ k_2 &= \mathsf{unsigned}\ (k_1 \cup k_2) \end{aligned}$$

$$\begin{pmatrix} \mathsf{signed}\ k_1 \cup \mathsf{unsigned}\ k_2 \\ \mathsf{unsigned}\ k_2 \cup \mathsf{signed}\ k_1 \end{pmatrix} = \begin{cases} \mathsf{unsigned}\ k_2 & \text{if } k_1 \subseteq k_2 \\ \mathsf{signed}\ k_1 & \text{if } k_1 \not\subseteq k_2 \text{ and } \mathsf{signed}\ k_1 \text{ can} \\ & \text{represent all } \mathsf{unsigned}\ k_2 \text{ values} \\ \mathsf{unsigned}\ k_1 & \text{otherwise} \end{cases}$$

We will now demonstrate the integer promotions and usual arithmetic conversions by means of a small example.

```
unsigned char x = 100, y = 3;
unsigned char z1 = (x * y) / y;                    /* 300 / 3 = 100 */
unsigned char z2 = (unsigned char)(x * y) / y;     /*  44 / 3 = 14  */
```

Assuming an implementation with $\mathsf{char\_bits} = 8$ and $\mathsf{rank\_size}\ \mathsf{int} = 4$, the arguments `x` and `y` are both promoted to $\lceil \mathsf{unsigned\ char} \rceil = \mathsf{signed\ int}$. The type of `x * y` is thereby $\mathsf{signed\ int} \cup \mathsf{signed\ int} = \mathsf{signed\ int}$, and its result is 300. The resulting value `z1` is thus $300/3 = 100$. In case the product is truncated to $\mathsf{unsigned\ char}$, as done in the assignment to `z2`, the result is $44/3 = 14$.

Definition 3.2.5. *An* integer environment with ranks K *extends an integer representation environment with the following evaluation functions:*

$$\begin{aligned}
\mathsf{int\_arithop\_ok} &: \mathsf{arithop} \to \mathbb{Z} \to \mathsf{inttype} \to \mathbb{Z} \to \mathsf{inttype} \to \mathsf{Prop}\\
\mathsf{int\_arithop} &: \mathsf{arithop} \to \mathbb{Z} \to \mathsf{inttype} \to \mathbb{Z} \to \mathsf{inttype} \to \mathbb{Z}\\
\mathsf{int\_shiftop\_ok} &: \mathsf{shiftop} \to \mathbb{Z} \to \mathsf{inttype} \to \mathbb{Z} \to \mathsf{inttype} \to \mathsf{Prop}\\
\mathsf{int\_shiftop} &: \mathsf{shiftop} \to \mathbb{Z} \to \mathsf{inttype} \to \mathbb{Z} \to \mathsf{inttype} \to \mathbb{Z}\\
\mathsf{int\_cast\_ok} &: \mathsf{inttype} \to \mathbb{Z} \to \mathsf{Prop}\\
\mathsf{int\_cast} &: \mathsf{inttype} \to \mathbb{Z} \to \mathbb{Z}
\end{aligned}$$

These functions should satisfy the following properties:

1. *The results should be well-typed, that is:*

$$\frac{x : \tau_\mathsf{i} \qquad y : \sigma_\mathsf{i} \qquad \mathsf{int\_arithop\_ok}\ \circledcirc_a\ x\ \tau_\mathsf{i}\ y\ \sigma_\mathsf{i}}{\mathsf{int\_arithop}\ \circledcirc_a\ x\ \tau_\mathsf{i}\ y\ \sigma_\mathsf{i} : \lceil \tau_\mathsf{i} \rceil \cup \lceil \sigma_\mathsf{i} \rceil}$$

$$\frac{x : \tau_\mathsf{i} \qquad y : \sigma_\mathsf{i} \qquad \mathsf{int\_shiftop\_ok}\ \circledcirc_a\ x\ \tau_\mathsf{i}\ y\ \sigma_\mathsf{i}}{\mathsf{int\_shiftop}\ \circledcirc_a\ x\ \tau_\mathsf{i}\ y\ \sigma_\mathsf{i} : \lceil \tau_\mathsf{i} \rceil} \qquad \frac{\mathsf{int\_cast\_ok}\ \sigma_\mathsf{i}\ x}{\mathsf{int\_cast}\ \sigma_\mathsf{i}\ x : \sigma_\mathsf{i}}$$

2. *If an arithmetic operation (respectively shift operation or cast) has defined behavior according to the C11 standard, the condition* $\mathsf{int\_arithop\_ok}\ \circledcirc_a\ x\ \tau_\mathsf{i}\ y\ \sigma_\mathsf{i}$ *(respectively* $\mathsf{int\_shiftop\_ok}\ \circledcirc_s\ x\ \tau_\mathsf{i}\ y\ \sigma_\mathsf{i}$ *or* $\mathsf{int\_cast\_ok}\ \tau_\mathsf{i}\ x$*) should hold.*
3. *If an arithmetic operation (respectively shift operation or cast) has defined behavior according to the C11 standard, the value* $\mathsf{int\_arithop}\ \circledcirc_a\ x\ \tau_\mathsf{i}\ y\ \sigma_\mathsf{i}$ *(respectively* $\mathsf{int\_shiftop}\ \circledcirc_s\ x\ \tau_\mathsf{i}\ y\ \sigma_\mathsf{i}$ *or* $\mathsf{int\_cast}\ \tau_\mathsf{i}\ x$*) should correspond to the mathematical value specified by C11.*

The precise statement of the last clauses is rather long winded and formally given as part of the Coq development.

The last two clauses of the previous definition seem in conflict with the goal of CH_2O to follow the C11 standard closely. This is not the case because CH_2O does not describe just one C semantics, but rather a *space* of possible C semantics. Each integer implementation corresponds to a point in this space, and among the points in this space, there are those that correspond to C11 and those that have fewer undefined behaviors than the C11 standard.

This space therefore also contains *de facto* standards such as the semantics of GCC or Clang with the `-fno-strict-overflow` flag enabled. This flag guarantees that signed integer overflow wraps around modulo instead of having undefined behavior. In C terminology, one may say that it is implementation-defined whether overflow of signed integer operators has undefined behavior in CH_2O.

3.3 Definition of types

CH_2O core C supports integer, pointer, function pointer, array, struct, union and void types. The translation from CH_2O abstract C into CH_2O core C translates more complicated types, such as typedefs and enums, into core types. This translation also alleviates other simplifications of the CH_2O core C type system, such as unnamed struct and union fields. Floating point types and qualifiers like `const` and `volatile` are not supported by CH_2O.

All definitions in this section are implicitly parameterized by an integer environment with ranks K (Definition 3.2.5).

Definition 3.3.1. *Tags* $t \in \mathsf{tag}$ *(sometimes called* struct/union names*) and* function names $f \in \mathsf{funname}$ *are represented as strings.*

Definition 3.3.2. *The* types of CH_2O core C*, collectively called* core types*, consist of* point-to types*,* base types *and* full types. *These are inductively defined as:*

$$\begin{aligned}
\tau_{\mathsf{p}}, \sigma_{\mathsf{p}} \in \mathsf{ptrtype} &::= \tau \mid \mathsf{any} \mid \vec{\tau} \to \tau \\
\tau_{\mathsf{b}}, \sigma_{\mathsf{b}} \in \mathsf{basetype} &::= \tau_{\mathsf{i}} \mid \tau_{\mathsf{p}}* \mid \mathsf{void} \\
\tau, \sigma \in \mathsf{type} &::= \tau_{\mathsf{b}} \mid \tau[n] \mid \mathsf{struct}\ t \mid \mathsf{union}\ t
\end{aligned}$$

The different kinds of types correspond to the different parts of the memory model. Addresses and pointers have point-to types (Definitions 5.2.7 on page 67 and 5.2.9 on page 68), base values have base types (Definition 5.5.2 on page 79), and memory trees and values have full types (Definitions 5.4.3 on page 5.4.3 and 5.5.8 on page 82).

As explained in Section 2.4.4 the void type of C is used for two entirely unrelated purposes: `void` is used for functions without return type and `void*` is used for pointers to objects of unspecified type. In CH_2O this distinction is explicit in the syntax of types. The type void is used for function without return value. Like the mathematical *unit* type it has one value called $\mathsf{nothing}$ (Definition 5.5.1 on page 79). The type $\mathsf{any}*$ is used for pointers to objects of unspecified type.

Unlike more modern programming languages C does not provide first class functions. Instead, C provides function pointers which are just addresses of executable code in memory instead of closures. Function pointers can be used in a way similar to ordinary pointers: they can be used as arguments and return value of functions, they can be part of structs, unions and arrays, *etc.*

The C language sometimes allows function types to be used as shorthands for function pointers, for example:

```
void sort(int *p, int len, int compare(int,int));
```

The third argument is a shorthand for `int (*compare)(int,int)` and is thus in fact a function pointer instead of a function. In CH$_2$O core C we only have function pointer types. The third argument of the core type of the function `sort` thus contains an additional $*$:

$$[\,(\mathsf{signed\ int})*,\ \mathsf{signed\ int},\ (\mathsf{signed\ int} \to \mathsf{signed\ int})*\,] \to \mathsf{void}.$$

In CH$_2$O core C struct and union types consist of just a name, and do not contain the types of their fields. An environment is used to assign fields to structs and unions, and to assign argument and return types to function names.

Definition 3.3.3. Type environments *are defined as:*

$$\begin{aligned} \Gamma \in \mathsf{env} := {} & (\mathsf{tag} \to_{\mathsf{fin}} \mathsf{list\ type}) \times && \text{(types of struct/union fields)} \\ & (\mathsf{funname} \to_{\mathsf{fin}} (\mathsf{list\ type} \times \mathsf{type})) && \text{(types of functions)} \end{aligned}$$

The functions $\mathsf{dom}_{\mathsf{tag}} : \mathsf{env} \to \mathcal{P}_{\mathsf{fin}}(\mathsf{tag})$ *and* $\mathsf{dom}_{\mathsf{funname}} : \mathsf{env} \to \mathcal{P}_{\mathsf{fin}}(\mathsf{funname})$ *yield the declared structs and unions, respectively the declared functions. We implicitly treat environments as functions* $\mathsf{tag} \to_{\mathsf{fin}} \mathsf{list\ type}$ *and* $\mathsf{funname} \to_{\mathsf{fin}} (\mathsf{list\ type} \times \mathsf{type})$ *that correspond to underlying finite partial functions.*

Struct and union names on the one hand, and function names on the other, have their own name space in accordance with the C11 standard [ISO12, 6.2.3p1].

Notation 3.3.4. *We often write an environment as a mixed sequence of struct and union declarations* $t : \vec{\tau}$*, and function declarations* $f : (\vec{\tau}, \tau)$*. This is possible because environments are finite.*

Since we represent the fields of structs and unions as lists in CH$_2$O core C, fields are nameless. For example, the C type `struct S1 { int x; struct S1 *p; }` is translated into the environment `S1` : $[\,\mathsf{signed\ int}, \mathsf{struct}\ \mathtt{S1}*\,]$ during the translation from CH$_2$O abstract C into CH$_2$O core C.

Although structs and unions are semantically very different (products versus sums, respectively), environments do not keep track of whether a tag has been used for a struct or a union type. Structs and union types with the same tag are thus allowed. The translator from CH$_2$O abstract C into CH$_2$O core C forbids the same name being used to declare both a struct and union type.

Although our mutual inductive syntax of types already forbids many incorrect types such as functions returning functions (instead of function pointers), still some ill-formed types such as `int[0]` are syntactically valid. Also, we have to ensure that cyclic structs and unions are only allowed when the recursive definition is guarded through pointers. Guardedness by pointers ensures that the sizes of types are finite and statically known. Consider the following types:

```
struct list1 { int hd; struct list1 tl; };     /* illegal */
struct list2 { int hd; struct list2 *p_tl; };  /* legal */
```

The type declaration `struct list1` is illegal because it has a reference to itself. In the type declaration `struct list2` the self reference is guarded through a pointer type, and therefore legal. Of course, this generalizes to mutual recursive types like:

```
struct tree { int hd; struct forest *p_children; };
struct forest { struct tree *p_hd; struct forest *p_tl; };
```

Definition 3.3.5. *The following judgments are defined by mutual induction:*

- *The judgment* $\Gamma \vdash_* \tau_{\mathsf{p}}$ *describes* point-to types τ_{p} to which a pointer may point*:*

$$\frac{}{\Gamma \vdash_* \mathsf{any}} \qquad \frac{\Gamma \vdash_* \vec{\tau} \qquad \Gamma \vdash_* \tau}{\Gamma \vdash_* \vec{\tau} \to \tau} \qquad \frac{\Gamma \vdash_{\mathsf{b}} \tau_{\mathsf{b}}}{\Gamma \vdash_* \tau_{\mathsf{b}}} \qquad \frac{\Gamma \vdash \tau \qquad n \neq 0}{\Gamma \vdash_* \tau[n]}$$

$$\frac{}{\Gamma \vdash_* \mathsf{struct}\ t} \qquad \frac{}{\Gamma \vdash_* \mathsf{union}\ t}$$

- *The judgment* $\Gamma \vdash_{\mathsf{b}} \tau_{\mathsf{b}}$ *describes* valid base types τ_{b}*:*

$$\frac{}{\Gamma \vdash_{\mathsf{b}} \tau_{\mathsf{i}}} \qquad \frac{\Gamma \vdash_* \tau_{\mathsf{p}}}{\Gamma \vdash_{\mathsf{b}} \tau_{\mathsf{p}}*} \qquad \frac{}{\Gamma \vdash_{\mathsf{b}} \mathsf{void}}$$

- *The judgment* $\Gamma \vdash \tau$ *describes* valid types τ*:*

$$\frac{\Gamma \vdash_{\mathsf{b}} \tau_{\mathsf{b}}}{\Gamma \vdash \tau_{\mathsf{b}}} \qquad \frac{\Gamma \vdash \tau \qquad n \neq 0}{\Gamma \vdash \tau[n]} \qquad \frac{t \in \mathsf{dom}_{\mathsf{tag}}\ \Gamma}{\Gamma \vdash \mathsf{struct}\ t} \qquad \frac{t \in \mathsf{dom}_{\mathsf{tag}}\ \Gamma}{\Gamma \vdash \mathsf{union}\ t}$$

Definition 3.3.6. *The judgment* $\vdash \Gamma$ *describes* well-formed environments Γ*. It is inductively defined as:*

$$\frac{}{\vdash \emptyset} \qquad \frac{\vdash \Gamma \quad \Gamma \vdash \vec{\tau} \quad \vec{\tau} \neq \varepsilon \quad t \notin \mathsf{dom}_{\mathsf{tag}}\ \Gamma}{\vdash t : \vec{\tau}, \Gamma} \qquad \frac{\vdash \Gamma \quad \Gamma \vdash_* \vec{\tau} \quad \Gamma \vdash_* \tau \quad f \notin \mathsf{dom}_{\mathsf{funname}}\ \Gamma}{\vdash f : (\vec{\tau}, \tau), \Gamma}$$

Note that $\Gamma \vdash \tau$ does not imply $\vdash \Gamma$. Most results therefore have $\vdash \Gamma$ as a premise. These premises are left implicit in this thesis.

In order to support (mutually) recursive struct and union types, pointers to incomplete struct and union types are permitted in the judgment $\Gamma \vdash_* \tau_{\mathsf{p}}$ that describes types to which pointers are allowed, but forbidden in the judgment $\Gamma \vdash \tau$ of validity of types. Let us consider the following type declarations:

```
struct S2 { struct S2 x; };   /* illegal */
struct S3 { struct S3 *p; };  /* legal */
```

Well-formedness $\vdash \Gamma$ of the environment $\Gamma := \mathtt{S3} : [\mathsf{struct}\ \mathtt{S3}*]$ can be derived using the judgments $\emptyset \vdash_* \mathsf{struct}\ \mathtt{S3}$, $\emptyset \vdash_{\mathsf{b}} \mathsf{struct}\ \mathtt{S3}*$, $\emptyset \vdash \mathsf{struct}\ \mathtt{S3}*$, and thus $\vdash \Gamma$. The environment $\mathtt{S2} : [\mathsf{struct}\ \mathtt{S2}]$ is ill-formed because we do not have $\emptyset \vdash \mathsf{struct}\ \mathtt{S2}$.

The typing rule for function pointers types is slightly more delicate. This is best illustrated by an example:

```
union U { int i; union U (*f) (union U); };
```

This example displays a recursive self reference to a union type through a function type, which is legal in C because function types are in fact pointer types. Due to this reason, the premises of $\Gamma \vdash_* \vec{\tau} \to \tau$ are $\Gamma \vdash_* \vec{\tau}$ and $\Gamma \vdash_* \tau$ instead of $\Gamma \vdash \vec{\tau}$ and $\Gamma \vdash \tau$. Well-formedness of the above union type can be derived as follows:

$$\dfrac{\vdash \Gamma \qquad \dfrac{\dfrac{}{\Gamma \vdash_{\mathsf{b}} \mathsf{signed\ int}}}{\Gamma \vdash \mathsf{signed\ int}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash_* \mathsf{union}\ \mathtt{U}} \qquad \dfrac{}{\Gamma \vdash_* \mathsf{union}\ \mathtt{U}}}{\Gamma \vdash_* \mathsf{union}\ \mathtt{U} \to \mathsf{union}\ \mathtt{U}}}{\Gamma \vdash_{\mathsf{b}} (\mathsf{union}\ \mathtt{U} \to \mathsf{union}\ \mathtt{U})*}}{\Gamma \vdash (\mathsf{union}\ \mathtt{U} \to \mathsf{union}\ \mathtt{U})*}}{\vdash \mathtt{U} : [\,\mathsf{signed\ int}, (\mathsf{union}\ \mathtt{U} \to \mathsf{union}\ \mathtt{U})*\,],\ \Gamma}$$

In order to define operations by recursion over the structure of well-formed types (see for example Definition 5.5.7 on page 81, which turns a sequence of bits into a value), we often need to perform recursive calls on the types of fields of structs and unions. In Coq we have defined a custom recursor and induction principle using well-founded recursion. In this chapter, we will use these implicitly.

Affeldt *et al.* [AM13, AS14] have formalized non-cyclicity of types using a complex constraint on paths through types. Our definition of validity of environments (Definition 3.3.6) follows the structure of type environments, and is therefore well-suited to implement the aforementioned recursor and induction principle.

Definition 3.3.7. *The judgment* $\mathsf{complete}_\Gamma\ \tau$ *describes that* a type τ is complete. *It is inductively defined as:*

$$\dfrac{}{\mathsf{complete}_\Gamma\ \tau_{\mathsf{b}}} \qquad \dfrac{}{\mathsf{complete}_\Gamma\ \tau[n]} \qquad \dfrac{t \in \mathsf{dom}_{\mathsf{tag}}\ \Gamma}{\mathsf{complete}_\Gamma\ (\mathsf{struct}\ t)} \qquad \dfrac{t \in \mathsf{dom}_{\mathsf{tag}}\ \Gamma}{\mathsf{complete}_\Gamma\ (\mathsf{union}\ t)}$$

Fact 3.3.8. *We have* $\Gamma \vdash \tau$ *iff* $\Gamma \vdash_* \tau$ *and* $\mathsf{complete}_\Gamma\ \tau$*.*

As discussed in Section 2.4.3, there is a close correspondence between array and pointer types in C. Arrays are not first class types, and except for special cases such as initialization, manipulation of arrays is achieved via pointers. In $\mathrm{CH_2O}$ core C we consider arrays as first class types so as to avoid having to make exceptions for the case of arrays all the time.

Due to this reason, more types are considered valid in $\mathrm{CH_2O}$ core C than considered valid in C11. The translator from $\mathrm{CH_2O}$ abstract C resolves exceptional cases for arrays. For example, a function parameter of array type acts like a parameter of pointer type in C11 [ISO12, 6.7.6.3][1].

```
void f(int a[10]);
```

The corresponding core type of `f` is thus $(\mathsf{signed\ int})* \to \mathsf{void}$. Note that the core type $(\mathsf{signed\ int})[10] \to \mathsf{void}$ is also valid, but entirely different, and never generated by the translator from $\mathrm{CH_2O}$ abstract C.

[1] The array size is ignored unless the `static` keyword is used. In case `f` would have the prototype `void f(int a[static 10])`, the pointer `a` should provide access to an array of at least 10 elements [ISO12, 6.7.6.3]. The `static` keyword is not supported by $\mathrm{CH_2O}$.

3.4 Implementation environments

We finish this chapter by extending integer environments to describe implementation-defined properties related the layout of struct and union types. Section 7.3 defines inhabitants of this interface corresponding to actual computing architectures.

Definition 3.4.1. *A* implementation environment with ranks K *consists of an integer environment with ranks K and functions:*

$$\mathsf{sizeof}_\Gamma : \mathsf{type} \to \mathbb{N} \qquad \mathsf{alignof}_\Gamma : \mathsf{type} \to \mathbb{N} \qquad \mathsf{fieldsizes}_\Gamma : \mathsf{list\ type} \to \mathsf{list}\ \mathbb{N}$$

These functions should satisfy:

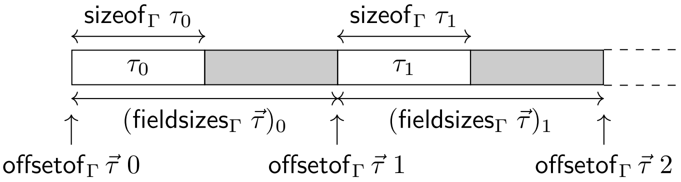
$$\begin{array}{rll}
\mathsf{sizeof}_\Gamma\ (si\ k) = \mathsf{rank\_size}\ k & \mathsf{sizeof}_\Gamma\ (\tau_\mathsf{p}*) \neq 0 & \mathsf{sizeof}_\Gamma\ \mathsf{void} \neq 0 \\
\mathsf{sizeof}_\Gamma\ (\tau[n]) = n * \mathsf{sizeof}_\Gamma\ \tau & & \\
\mathsf{sizeof}_\Gamma\ (\mathsf{struct}\ t) = \Sigma\ \mathsf{fieldsizes}_\Gamma\ \vec{\tau} & \text{if } \Gamma\, t = \vec{\tau} & \\
\mathsf{sizeof}_\Gamma\ \tau_i \leq z_i \text{ and } |\vec{\tau}| = |\vec{z}| & \text{if } \mathsf{fieldsizes}_\Gamma\ \vec{\tau} = \vec{z}, \text{ for each } i < |\vec{\tau}| & \\
\mathsf{sizeof}_\Gamma\ \tau_i \leq \mathsf{sizeof}_\Gamma\ (\mathsf{union}\ t) & \text{if } \Gamma\, t = \vec{\tau}, \text{ for each } i < |\vec{\tau}| & \\
\mathsf{alignof}_\Gamma\ \tau \mid \mathsf{alignof}_\Gamma\ (\tau[n]) & & \\
\mathsf{alignof}_\Gamma\ \tau_i \mid \mathsf{alignof}_\Gamma\ (\mathsf{struct}\ t) & \text{if } \Gamma\, t = \vec{\tau}, \text{ for each } i < |\vec{\tau}| & \\
\mathsf{alignof}_\Gamma\ \tau_i \mid \mathsf{alignof}_\Gamma\ (\mathsf{union}\ t) & \text{if } \Gamma\, t = \vec{\tau}, \text{ for each } i < |\vec{\tau}| & \\
\mathsf{alignof}_\Gamma\ \tau \mid \mathsf{sizeof}_\Gamma\ \tau & \text{if } \Gamma \vdash \tau & \\
\mathsf{alignof}_\Gamma\ \tau_i \mid \mathsf{offsetof}_\Gamma\ \vec{\tau}\ i & \text{if } \Gamma \vdash \vec{\tau}, \text{ for each } i < |\vec{\tau}| &
\end{array}$$

Here, we let $\mathsf{offsetof}_\Gamma\ \vec{\tau}\ i$ *denote* $\Sigma_{j<i}(\mathsf{fieldsizes}_\Gamma\ \vec{\tau})_j$. *The functions* sizeof_Γ, $\mathsf{alignof}_\Gamma$, *and* $\mathsf{fieldsizes}_\Gamma$ *should be closed under weakening of* Γ.

Notation 3.4.2. *Given an implementation environment, we let:*

$$\begin{aligned}
\mathsf{bitsizeof}_\Gamma\ \tau &:= \mathsf{sizeof}_\Gamma\ \tau \cdot \mathsf{char\_bits} \\
\mathsf{bitoffsetof}_\Gamma\ \tau\ j &:= \mathsf{offsetof}_\Gamma\ \tau\ j \cdot \mathsf{char\_bits} \\
\mathsf{fieldbitsizes}_\Gamma\ \tau &:= \mathsf{fieldsizes}_\Gamma\ \tau \cdot \mathsf{char\_bits}
\end{aligned}$$

We let $\mathsf{sizeof}_\Gamma\ \tau$ specify the number of bytes out of which the object representation of an object of type τ constitutes. Objects of type τ should be allocated at addresses that are a multiple of $\mathsf{alignof}_\Gamma\ \tau$. We will prove that our abstract notion of addresses satisfies this property (see Lemma 5.2.17 on page 70). The functions sizeof_Γ, $\mathsf{alignof}_\Gamma$ correspond to the `sizeof` and `_Alignof` operators [ISO12, 6.5.3.4], and $\mathsf{offsetof}_\Gamma$ corresponds to the `offsetof` macro [ISO12, 7.19p3]. The list $\mathsf{fieldsizes}_\Gamma\ \vec{\tau}$ specifies the layout of a struct type with fields $\vec{\tau}$ as follows:



$\mathsf{sizeof}_\Gamma\ \tau_0$
$\mathsf{sizeof}_\Gamma\ \tau_1$
τ_0
τ_1
$(\mathsf{fieldsizes}_\Gamma\ \vec{\tau})_0$
$(\mathsf{fieldsizes}_\Gamma\ \vec{\tau})_1$
$\mathsf{offsetof}_\Gamma\ \vec{\tau}\ 0$
$\mathsf{offsetof}_\Gamma\ \vec{\tau}\ 1$
$\mathsf{offsetof}_\Gamma\ \vec{\tau}\ 2$

Chapter 4

Permissions and separation algebras

Permissions control whether memory operations such as a read or store are allowed or not. In order to obtain the highest level of precision, we tag each individual bit in memory with a corresponding permission. In the operational semantics, permissions have two main purposes:

- Permissions are used to formalize the *sequence point restriction* which assigns undefined behavior to programs in which an object in memory is modified more than once in between two sequence points.
- Permissions are used to distinguish objects in memory that are writable from those that are read-only (*const qualified* in C terminology).

In the axiomatic semantics based on separation logic, permissions play an important role for *share accounting.* We use share accounting for *subdivision of permissions* among multiple subexpressions to ensure that:

- Writable objects are unique to each subexpression.
- Read-only objects may be shared between subexpressions.

This distinction is originally due to Dijkstra [Dij68] and is essential in separation logic with permissions [BCOP05]. The novelty of our work is to use separation logic with permissions for non-determinism in expressions in C. Share accounting gives rise to a natural treatment of C's sequence point restriction.

Separation algebras as introduced by Calcagno *et al.* [COY07] abstractly capture common structure of subdivision of permissions. We present a generalization of separation algebras that is well-suited for C verification in Coq and use this generalization to build the permission system and memory model compositionally. The permission system will be constructed as a telescope of separation algebras:

$$\mathsf{perm} \quad := \quad \underbrace{\mathcal{L}(\mathcal{C}(\mathbb{Q}))}_{\text{non-const qualified}} \quad + \quad \underbrace{\mathbb{Q}}_{\text{const qualified}}$$

Here, $\mathbb{Q}$ is the separation algebra of fractional permissions, $\mathcal{C}$ is a functor that extends a separation algebra with a counting component, and $\mathcal{L}$ is a functor that extends a separation algebra with a lockable component (used for the sequence point restriction). This chapter explains these functors and their purposes in detail.

4.1 Separation logic and share accounting

Before we will go into the details of the $\mathrm{CH_2O}$ permission system, we briefly introduce separation logic. Separation logic [ORY01] is an extension of Hoare logic that provides better means to reason about imperative programs that use mutable data structures and pointers. The key feature of separation logic is the *separating conjunction* $P * Q$ that allows one to subdivide the memory into two disjoint parts: a part described by P and another part described by Q. The separating conjunction is most prominent in the *frame rule*.

$$\frac{\{P\}\, s\, \{Q\}}{\{P * R\}\, s\, \{Q * R\}}$$

This rule enables local reasoning. Given a Hoare triple $\{P\}\, s\, \{Q\}$, this rule allows one to derive that the triple also holds when the memory is extended with a disjoint part described by R. The frame rule shows its merits when reasoning about functions. There it allows one to consider a function in the context of the memory the function actually uses, instead of having to consider the function in the context of the entire program's memory. However, already in derivations of small programs the use of the frame rule can be demonstrated[1]:

$$\dfrac{\dfrac{\dfrac{}{\{\texttt{x} \mapsto 0\}\, \texttt{x:=10}\, \{\texttt{x} \mapsto 10\}}}{\{\texttt{x} \mapsto 0 * \texttt{y} \mapsto 0\}\, \texttt{x:=10}\, \{\texttt{x} \mapsto 10 * \texttt{y} \mapsto 0\}} \qquad \dfrac{\dfrac{}{\{\texttt{y} \mapsto 0\}\, \texttt{y:=12}\, \{\texttt{y} \mapsto 12\}}}{\{\texttt{x} \mapsto 10 * \texttt{y} \mapsto 0\}\, \texttt{y:=12}\, \{\texttt{x} \mapsto 10 * \texttt{y} \mapsto 12\}}}{\{\texttt{x} \mapsto 0 * \texttt{y} \mapsto 0\}\, \texttt{x:=10;\ y:=12}\, \{\texttt{x} \mapsto 10 * \texttt{y} \mapsto 12\}}$$

The *singleton assertion* $a \mapsto v$ denotes that the memory consists of exactly one object with value v at address a. The assignments are not considered in the context of the entire memory, but just in the part of the memory that is used.

The key observation that led to our separation logic for C as presented in Chapter 8 is the correspondence between non-determinism in expressions and a form of concurrency. Inspired by the rule for the parallel composition [O'H04], we have rules for each operator $\circledcirc$ that are of the following shape.

$$\frac{\{P_1\}\, e_1\, \{Q_1\} \qquad \{P_2\}\, e_2\, \{Q_2\}}{\{P_1 * P_2\}\, e_1 \circledcirc e_2\, \{Q_1 * Q_2\}}$$

The intuitive idea of this rule is that if the memory can be subdivided into two parts in which the subexpressions e_1 and e_2 can be executed safely, then the expression $e_1 \circledcirc e_2$ can be executed safely in the whole memory. Non-interference of the side-effects of e_1 and e_2 is guaranteed by the separating conjunction. It ensures that the parts of the memory described by P_1 and P_2 do not have overlapping areas that will be written to. We thus effectively rule out expressions with undefined behavior such as `(x = 3) + (x = 4)` (see Section 2.5.9 for discussion).

Subdividing the memory into multiple parts is not a simple operation. In order to illustrate this, let us consider a shallow embedding of assertions of separation logic

[1] Contrary to traditional separation logic, we do not give local variables a special status of being *stack allocated*. We do so, because in C even local variables are allowed to have pointers to them. See Section 2.5.1 for an example.

$P, Q : \mathsf{mem} \to \mathsf{Prop}$ (think of mem as being the set of finite partial functions from some set of object identifiers to some set of objects. The exact definition in the context of CH_2O is given in Definition 5.4.4 on page 74). In such a shallow embedding, one would define the separating conjunction as follows:

$$P * Q := \lambda m \,.\, \exists m_1\, m_2 \,.\, m = m_1 \cup m_2 \wedge P\, m_1 \wedge Q\, m_2.$$

The operation $\cup$ is *not* the disjoint union of finite partial functions, but a more fine grained operation. There are two reasons for that. Firstly, subdivision of memories should allow for partial overlap, as long as writable objects are unique to a single part. For example, the expression `x + x` has defined behavior, but the expressions `x + (x = 4)` and `(x = 3) + (x = 4)` have not.

We use separation logic with permissions [BCOP05] to deal with partial overlap of memories. That means, we equip the singleton assertion $a \overset{\gamma}{\mapsto} v$ with a permission γ. The essential property of the singleton assertion is that given a writable permission γ_w there is a readable permission γ_r with:

$$(a \overset{\gamma_w}{\longmapsto} v) \quad \leftrightarrow \quad (a \overset{\gamma_r}{\longmapsto} v) * (a \overset{\gamma_r}{\longmapsto} v).$$

The above property is an instance of a slightly more general property. We consider a binary operation $\cup$ on permissions so we can write (see Lemma 8.3.6 on page 154 for the actual result):

$$(a \overset{\gamma_1 \cup \gamma_2}{\longmapsto} v) \quad \leftrightarrow \quad (a \overset{\gamma_1}{\longmapsto} v) * (a \overset{\gamma_2}{\longmapsto} v).$$

Secondly, it should be possible to subdivide array, struct and union objects into subobjects corresponding to their elements. For example, in the case of an array `int a[2]`, the expression `(a[0] = 1) + (a[1] = 4)` has defined behavior, and we should be able to prove so. The essential property of the singleton assertion for an $\mathsf{array}\,[\,y_0, \ldots, y_{n-1}\,]$ value is (see Lemma 8.3.5 on page 153 for the actual result):

$$(a \overset{\gamma}{\mapsto} \mathsf{array}\,[\,v_0, \ldots, v_{n-1}\,]) \quad \leftrightarrow \quad (a[0] \overset{\gamma}{\mapsto} v_0) * \cdots * (a[n-1] \overset{\gamma}{\mapsto} v_{n-1}).$$

4.2 Separation algebras

As shown in the previous section, the key operation needed to define a shallow embedding of separation logic with permissions is a binary operation $\cup$ on memories and permissions. Calcagno *et al.* introduced the notion of a *separation algebra* [COY07] so as to capture common properties of the $\cup$ operation. A *separation algebra* $(A, \emptyset, \cup)$ is a partial cancellative commutative monoid (see Definition 4.2.1 for our actual definition). Some prototypical instances of separation algebras are:

- Finite partial functions $(A \to_{\mathsf{fin}} B, \emptyset, \cup)$, where $\emptyset$ is the empty finite partial function, and $\cup$ the disjoint union on finite partial functions.
- The Booleans $(\mathsf{bool}, \mathsf{false}, \vee)$.
- Boyland's fractional permissions $([0,1]_{\mathbb{Q}}, 0, +)$ where 0 denotes no access, 1 denotes writable access, and $0 < \_ < 1$ denotes read-only access [BCOP05, Boy03].

Separation algebras are also closed under various constructs (such as products and finite functions), and complex instances can thus be built compositionally.

When formalizing separation algebras in the Coq proof assistant, we quickly ran into some problems:

- Dealing with partial operations such as $\cup$ is cumbersome, see Section 9.5.
- Dealing with subset types (modeled as Σ-types) is inconvenient.
- Operations such as the difference operation $\setminus$ cannot be defined constructively from the laws of a separation algebra.

In order to deal with the issue of partiality, we turn $\cup$ into a total operation. Only in case x and y are *disjoint*, notation $x \perp y$, we require $x \cup y$ to satisfy the laws of a separation algebra. Instead of using subsets, we equip separation algebras with a predicate $\mathsf{valid} : A \to \mathsf{Prop}$ that explicitly describes a subset of the carrier A. Lastly, we explicitly add a difference operation $\setminus$.

Definition 4.2.1. *A* separation algebra *consists of a type A, with:*

- *An element* $\emptyset : A$
- *A predicate* $\mathsf{valid} : A \to \mathsf{Prop}$
- *Binary relations* $\perp,\ \subseteq\ : A \to A \to \mathsf{Prop}$
- *Binary operations* $\cup,\ \setminus\ : A \to A \to A$

Satisfying the following laws:

1. *If* $\mathsf{valid}\ x$, *then* $\emptyset \perp x$ *and* $\emptyset \cup x = x$
2. *If* $x \perp y$, *then* $y \perp x$ *and* $x \cup y = y \cup x$
3. *If* $x \perp y$ *and* $x \cup y \perp z$, *then* $y \perp z$, $x \perp y \cup z$, *and* $x \cup (y \cup z) = (x \cup y) \cup z$
4. *If* $z \perp x$, $z \perp y$ *and* $z \cup x = z \cup y$, *then* $x = y$
5. *If* $x \perp y$, *then* $\mathsf{valid}\ x$ *and* $\mathsf{valid}\ (x \cup y)$
6. *If* $x \perp y$ *and* $x \cup y = \emptyset$, *then* $x = \emptyset$
7. *If* $x \perp y$, *then* $x \subseteq x \cup y$
8. *If* $x \subseteq y$, *then* $x \perp y \setminus x$ *and* $x \cup y \setminus x = y$

Laws 1–4 describe the traditional laws of a separation algebra: identity, commutativity, associativity and cancellativity. Law 5 ensures that valid is closed under the $\cup$ operation. Law 6 describes positivity. Laws 7 and 8 fully axiomatize the $\subseteq$ relation and $\setminus$ operation. Using the positivity and cancellation law, we obtain that $\subseteq$ is a partial order in which $\cup$ is order preserving and respecting.

In case of permissions, the $\emptyset$ element is used to split objects of compound types (arrays and structs) into multiple parts as detailed in Section 8.3. To that end, we use separation algebras instead of *permission algebras* [ORY01], which are a variant of separation algebras without an $\emptyset$ element.

Definition 4.2.2. *The* Boolean separation algebra bool *is defined as:*

$$\begin{aligned} \mathsf{valid}\ x &:= \mathsf{True} & \emptyset &:= \mathsf{false} \\ x \perp y &:= \neg x \vee \neg y & x \cup y &:= x \vee y \\ x \subseteq y &:= x \to y & x \setminus y &:= x \wedge \neg y \end{aligned}$$

In the case of fractional permissions $[0,1]_{\mathbb{Q}}$ the problem of partiality and subset types already clearly appears. The $\cup$ operation (here $+$) can 'overflow'. We remedy this problem by having all operations operate on pre-terms (here $\mathbb{Q}$) and the predicate valid describes validity of pre-terms (here $0 \leq \_ \leq 1$).

Definition 4.2.3. *The* fractional separation algebra $\mathbb{Q}$ *is defined as:*

$$\begin{aligned} \mathsf{valid}\ x &:= 0 \leq x \leq 1 & \emptyset &:= 0 \\ x \perp y &:= 0 \leq x, y \ \wedge\ x + y \leq 1 & x \cup y &:= x + y \\ x \subseteq y &:= 0 \leq x \leq y \leq 1 & x \setminus y &:= x - y \end{aligned}$$

The version of separation algebras by Klein *et al.* [KKB12] in Isabelle also models $\cup$ as a total operation and uses a relation $\perp$. There are some differences:

- We include a predicate valid to prevent having to deal with subset types.
- They have weaker premises for associativity (law 3), namely $x \perp y$, $y \perp z$ and $x \perp z$ instead of $x \perp y$ and $x \cup y \perp z$. Ours are more natural, *e.g.* for fractional permissions one has $0.5 \perp 0.5$ but not $0.5 + 0.5 \perp 0.5$, and it thus makes no sense to require $0.5 \cup (0.5 \cup 0.5) = (0.5 \cup 0.5) \cup 0.5$ to hold.
- Since Coq (without axioms) does not have a choice operator, the $\setminus$ operation cannot be defined in terms of $\cup$. Isabelle has a choice operator.

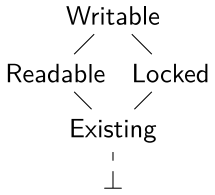
Dockins *et al.* [DHA09] have formalized a hierarchy of different separation algebras in Coq. They have dealt with the issue of partiality by treating $\cup$ as a relation instead of a function. This is unnatural, because equational reasoning becomes impossible and one has to name all auxiliary results.

Bengtson *et al.* [BJSB11] have formalized separation algebras in Coq to reason about object-oriented programs. They have treated $\cup$ as a partial function, and have not defined any complex permission systems.

4.3 Permissions

We classify permissions using *permission kinds.*

Definition 4.3.1. *The lattice of* permission kinds $(\mathsf{pkind}, \subseteq)$ *is defined as:*



The order $k_1 \subseteq k_2$ expresses that k_1 allows fewer operations than k_2. This organization of permissions is inspired by that of Leroy *et al.* [LABS12]. The intuitive meaning of the permission kinds is as follows:

- Writable. *Writable permissions* allow reading and writing.

- Readable. *Read-only permissions* allow solely reading.
- Existing. *Existence permissions* [BCOP05] are used for objects that are known to exist but whose value cannot be used. Existence permissions are essential because C only permits pointer arithmetic on pointers that refer to objects that have not been deallocated (see Section 2.5.7 for discussion).
- Locked. *Locked permissions* are used to formalize the sequence point restriction. When an object is modified during the execution of an expression, it is temporarily given a locked permission to forbid any read/write accesses until the next sequence point. Section 6.4 details the treatment of sequence points in the operational semantics.

  For example, in `(x = 3) + *p;` the assignment `x = 3` locks the permissions of the object `x`. Since future read/write accesses to `x` are forbidden, accessing `*p` results in undefined in case `p` points to `x`. At the sequence point `;`, the original permission of `x` is restored.

  Locked permissions are different from existence permissions because the operational semantics can change writable permissions into locked permissions and *vice versa*, but cannot do that with existence permissions.
- $\bot$. *Empty permissions* allow no operations.

In our separation logic we do not only have control which operations are allowed, but also have to deal with share accounting.

- We need to subdivide objects with writable or read-only permission into multiple parts with read-only permission. For example, in the expression `x + x`, both subexpressions require `x` to have at least read-only permission.
- We need to subdivide objects with writable permission into a part with existence permission and a part with writable permission. For example, in the expression `*(p + 1) = (*p = 1)`, the subexpression `*p = 1` requires `*p` to have writable permission, and the subexpression `*(p + 1)` requires `*p` to have at least existence permission so as to perform pointer arithmetic on `p`.

When reassembling subdivided permissions (using $\cup$), we need to know when the original permission is reobtained. Therefore, the underlying permission system needs to have more structure, and cannot consist of just the permission kinds.

Definition 4.3.2. CH$_2$O permissions perm *are defined as:*

$$\gamma \in \mathsf{perm} \quad := \quad \underbrace{\overset{\textit{Lockable SA}}{\mathcal{L}}(\overset{\textit{Counting SA}}{\mathcal{C}}(\overset{\textit{Fractional SA}}{\mathbb{Q}}))}_{\text{non-const qualified}} \qquad + \qquad \underbrace{\overset{\textit{Fractional SA}}{\mathbb{Q}}}_{\text{const qualified}}$$

where $\mathcal{L}(A) := \{\blacklozenge, \lozenge\} \times A$ *and* $\mathcal{C}(A) := \mathbb{Q} \times A$.

In Section 4.5 we will give the exact definition of the separation algebra operations on permissions by defining these one by one for the counting separation algebra $\mathcal{C}$, the

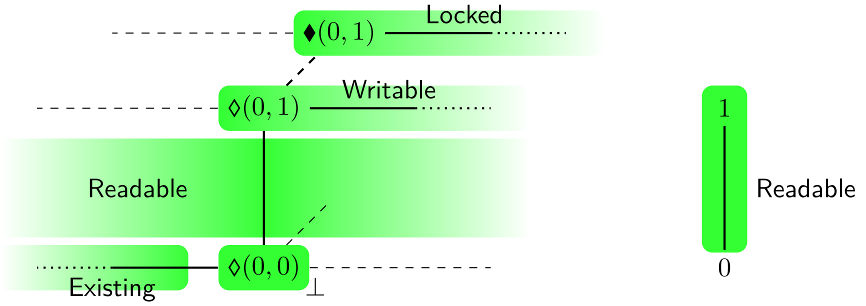


Figure 4.1: The CH_2O permission system.

lockable separation algebra $\mathcal{L}$, and the separation algebra on sums $+$. This section gives a summary of the important aspects of the permission system.

We combine fractional permissions to account for read-only/writable permissions with counting permissions to account for the number of existence permissions that have been handed out. The annotations $\{\blacklozenge, \lozenge\}$ describe whether a permission is locked $\blacklozenge$ or not $\lozenge$. Only writable permissions have a locked variant.

Const permissions are used for objects declared with the `const` qualifier (currently we only support string literals that are implicitly declared const). Modifying an object with const permissions results in undefined behavior. Const permissions do not have a locked variant or a counting component because they do not allow writing.

Figure 4.1 indicates the valid predicate by the areas marked green and displays how the elements of the permission system project onto their kinds. The operation $\cup$ is defined roughly as the point-wise addition and $\setminus$ as point-wise subtraction.

We will define an operation $\frac{1}{2}$: $\mathsf{perm} \to \mathsf{perm}$ to subdivide a writable or read-only permission into read-only permissions.

$$\tfrac{1}{2}\gamma := \begin{cases} \lozenge(0.5 \cdot x,\, 0.5 \cdot y) & \text{if } \gamma = \lozenge(x,\, y) \\ 0.5 \cdot x & \text{if } \gamma = x \in \mathbb{Q} \\ \gamma & \text{otherwise, dummy value} \end{cases}$$

Given a writable or read-only permission γ, the subdivided read-only permission $\frac{1}{2}\gamma$ enjoys $\frac{1}{2}\gamma \perp \frac{1}{2}\gamma$ and $\frac{1}{2}\gamma \cup \frac{1}{2}\gamma = \gamma$.

The existence permission $\mathsf{token} := \lozenge(-1,\, 0)$ is used in combination with the $\setminus$ operation to subdivide a writable permission γ into a writable permission $\gamma \setminus \mathsf{token}$ and an existence permission token. We have $\mathsf{token} \cup (\gamma \setminus \mathsf{token}) = \gamma$ by law 8 of separation algebras. Importantly, only objects with $\lozenge(0,\, 1)$ permission can be deallocated, whereas objects with $\gamma \setminus \mathsf{token}$ permission cannot (see Definition 5.6.5 on page 85) because expressions such as `(p == p) + (free(p),0)` have undefined behavior.

4.4 Extended separation algebras

We extend separation algebras with a split operation $\frac{1}{2}$ and predicates to distinguish permissions in our memory model.

Definition 4.4.1. *An* extended separation algebra *extends a separation algebra with:*

- *Predicates* $\mathsf{splittable}, \mathsf{unmapped}, \mathsf{exclusive} : A \to \mathsf{Prop}$
- *A unary operation* $\frac{1}{2} : A \to A$

Satisfying the following laws:

9. *If* $x \perp x$*, then* $\mathsf{splittable}\ (x \cup x)$
10. *If* $\mathsf{splittable}\ x$*, then* $\frac{1}{2}x \perp \frac{1}{2}x$ *and* $\frac{1}{2}x \cup \frac{1}{2}x = x$
11. *If* $\mathsf{splittable}\ y$ *and* $x \subseteq y$*, then* $\mathsf{splittable}\ x$
12. *If* $x \perp y$ *and* $\mathsf{splittable}\ (x \cup y)$*, then* $\frac{1}{2}(x \cup y) = \frac{1}{2}x \cup \frac{1}{2}y$
13. $\mathsf{unmapped}\ \emptyset$*, and if* $\mathsf{unmapped}\ x$*, then* $\mathsf{valid}\ x$
14. *If* $\mathsf{unmapped}\ y$ *and* $x \subseteq y$*, then* $\mathsf{unmapped}\ x$
15. *If* $x \perp y$*,* $\mathsf{unmapped}\ x$ *and* $\mathsf{unmapped}\ y$*, then* $\mathsf{unmapped}\ (x \cup y)$
16. $\mathsf{exclusive}\ x$ *iff* $\mathsf{valid}\ x$ *and for all* y *with* $x \perp y$ *we have* $\mathsf{unmapped}\ y$
17. *Not both* $\mathsf{exclusive}\ x$ *and* $\mathsf{unmapped}\ x$
18. *There exists an* x *with* $\mathsf{valid}\ x$ *and not* $\mathsf{unmapped}\ x$

The $\frac{1}{2}$-operation is partial, but described by a total function whose result $\frac{1}{2}x$ is only meaningful if $\mathsf{splittable}\ x$ holds. Law 11 ensures that splittable permissions are infinitely splittable, and law 12 ensures that $\frac{1}{2}$ distributes over $\cup$.

The predicates $\mathsf{unmapped}$ and $\mathsf{exclusive}$ associate an intended semantics to elements of a separation algebra. Let us consider fractional permissions to indicate the intended meaning of these predicates.

Definition 4.4.2. *The* fractional separation algebra $\mathbb{Q}$ *is extended with:*

$$\mathsf{splittable}\ x := 0 \le x \le 1 \qquad \tfrac{1}{2}x := 0.5 \cdot x$$
$$\mathsf{unmapped}\ x := x = 0 \qquad \mathsf{exclusive}\ x := x = 1$$

Remember that permissions will be used to annotate each individual bit in memory. Unmapped permissions are *on the bottom*: they do not allow their bit to be used in any way. Exclusive permissions are *on the top*: they are the sole owner of a bit and can do anything to that bit without affecting disjoint bits.

Fractional permissions have exactly one unmapped element and exactly one exclusive element, but in the CH$_2$O permission system this is not the case. The elements of the CH$_2$O permission system are classified as follows:

unmapped	exclusive	Examples
	✓	Writable and Locked permissions
		Readable permissions
✓		The $\emptyset$ permission and Existing permissions

In order to abstractly describe bits annotated with permissions we define the tagged separation algebra $\mathcal{T}^t_T(A)$. In its concrete use $\mathcal{T}^{\lightning}_{\mathsf{bit}}(\mathsf{perm})$ in the memory model (Definition 5.3.3 on page 71), the elements (γ, b) consist of a permission $\gamma \in \mathsf{perm}$ and bit $b \in \mathsf{bit}$. We use the symbolic bit $\lightning$ that represents indeterminate storage to ensure that bits with $\mathsf{unmapped}$ permissions indeed have no usable value.

Definition 4.4.3. *Given a separation algebra A and a set of tags T with default tag $t \in T$, the* tagged separation algebra $\mathcal{T}_T^t(A) := A \times T$ over A *is defined as:*

$$\begin{aligned}
\mathsf{valid}\,(x,\,y) &:= \mathsf{valid}\ x \wedge (\mathsf{unmapped}\ x \to y = t)\\
\emptyset &:= (\emptyset,\,t)\\
(x,\,y) \perp (x',\,y') &:= x \perp x'\ \wedge (\mathsf{unmapped}\ x \vee y = y' \vee \mathsf{unmapped}\ x')\\
&\qquad\quad\ \wedge (\mathsf{unmapped}\ x \to y = t) \wedge (\mathsf{unmapped}\ x' \to y' = t)\\
(x,\,y) \cup (x',\,y') &:= \begin{cases}(x \cup x',\,y') & \text{if } y = t\\ (x \cup x',\,y) & \text{otherwise}\end{cases}\\
\mathsf{splittable}\,(x,\,y) &:= \mathsf{splittable}\ x \wedge (\mathsf{unmapped}\ x \to y = t)\\
\tfrac{1}{2}(x,\,y) &:= (\tfrac{1}{2}x,\,y)\\
\mathsf{unmapped}\,(x,\,y) &:= \mathsf{unmapped}\ x \wedge y = t\\
\mathsf{exclusive}\,(x,\,y) &:= \mathsf{exclusive}\ x
\end{aligned}$$

The definitions of the omitted relations and operations are as expected.

4.5 Permissions as a separation algebra

This section gives the full definitions of the separation algebras involved in the construction of the CH$_2$O permission system $\mathsf{perm} := \mathcal{L}(\mathcal{C}(\mathbb{Q})) + \mathbb{Q}$.

The *counting separation algebra over a separation algebra A* has elements $(x,\,y)$ with $x \in \mathbb{Q}$ and $y \in A$. The rational number x counts the number of existence permissions (*i.e.* permissions that only allow pointer arithmetic) that have been handed out. Existence permissions themselves are elements $(x,\,\emptyset)$ with $x < 0$. To ensure that the counting separation algebra is closed under $\cup$ and preserves splittability, the counter x is a rational number instead of an integer.

Definition 4.5.1. *Given a separation algebra A, the* counting separation algebra $\mathcal{C}(A) := \mathbb{Q} \times A$ over A *is defined as:*

$$\begin{aligned}
\mathsf{valid}\,(x,\,y) &:= \mathsf{valid}\ y \wedge (\mathsf{unmapped}\ y \to x \le 0) \wedge (\mathsf{exclusive}\ y \to 0 \le x)\\
\emptyset &:= (0,\,\emptyset)\\
(x,\,y) \perp (x',\,y') &:= y \perp y'\ \wedge (\mathsf{unmapped}\ y \to x \le 0) \wedge (\mathsf{unmapped}\ y' \to x' \le 0)\\
&\qquad\quad\ \wedge (\mathsf{exclusive}\ (y \cup y') \to 0 \le x + x')\\
(x,\,y) \cup (x',\,y') &:= (x + x',\,y \cup y')\\
\mathsf{splittable}\,(x,\,y) &:= \mathsf{splittable}\ y \wedge (\mathsf{unmapped}\ y \to x \le 0) \wedge (\mathsf{exclusive}\ y \to 0 \le x)\\
\tfrac{1}{2}(x,\,y) &:= (0.5 \cdot x,\,\tfrac{1}{2}y)\\
\mathsf{unmapped}\,(x,\,y) &:= \mathsf{unmapped}\ y \wedge x \le 0\\
\mathsf{exclusive}\,(x,\,y) &:= \mathsf{exclusive}\ y \wedge 0 \le x
\end{aligned}$$

The *lockable separation algebra* adds annotations $\{\blacklozenge, \lozenge\}$ to account for whether a permission is locked $\blacklozenge$ or not $\lozenge$. Permissions that are locked have exclusive write ownership, and are thus only disjoint from those that are $\mathsf{unmapped}$.

Definition 4.5.2. *Given a separation algebra A, the* lockable separation algebra $\mathcal{L}(A) := \{\blacklozenge, \lozenge\} \times A$ over A *is defined as:*

$$\begin{array}{rlrl}
\mathsf{valid}\ (\lozenge\, x) := & \mathsf{valid}\ x & \mathsf{valid}\ (\blacklozenge\, x) := & \mathsf{exclusive}\ x \\
\emptyset := & \lozenge\, \emptyset & & \\
\lozenge\, x \perp \lozenge\, y := & x \perp y & \lozenge\, x \perp \blacklozenge\, y := & x \perp y \wedge \mathsf{unmapped}\ x \wedge \mathsf{exclusive}\ y \\
\blacklozenge\, x \perp \blacklozenge\, y := & \mathsf{False} & \blacklozenge\, x \perp \lozenge\, y := & x \perp y \wedge \mathsf{exclusive}\ x \wedge \mathsf{unmapped}\ y \\
\lozenge\, x \cup \lozenge\, y := & \lozenge(x \cup y) & \lozenge\, x \cup \blacklozenge\, y := & \blacklozenge(x \cup y) \\
 & & \blacklozenge\, x \cup \lozenge\, y := & \blacklozenge(x \cup y) \\
\mathsf{splittable}\ (\blacklozenge\, x) := & \mathsf{False} & \mathsf{splittable}\ (\lozenge\, x) := & \mathsf{splittable}\ x \\
 & & \tfrac{1}{2}(\lozenge\, x) := & \lozenge(\tfrac{1}{2}x) \\
\mathsf{unmapped}\ (\blacklozenge\, x) := & \mathsf{False} & \mathsf{unmapped}\ (\lozenge\, x) := & \mathsf{unmapped}\ x \\
\mathsf{exclusive}\ (\blacklozenge\, x) := & \mathsf{exclusive}\ x & \mathsf{exclusive}\ (\blacklozenge\, x) := & \mathsf{exclusive}\ x
\end{array}$$

The separation algebra structure on the sum is intuitively straightforward, but technically subtle because the $\emptyset$ elements have to be *identified.* We achieve this by ensuring that only the $\emptyset$ element of A in $A + B$ is used.

Definition 4.5.3. *Given a separation algebras A and B, the separation algebra on the sum $A + B$ is defined as:*

$$\begin{array}{rlrl}
\mathsf{valid}\ x_{\mathbf{l}} := & \mathsf{valid}\ x & \mathsf{valid}\ x_{\mathbf{r}} := & \mathsf{valid}\ x \wedge x \neq \emptyset \\
\emptyset := & \emptyset_{\mathbf{l}} & & \\
x_{\mathbf{l}} \perp y_{\mathbf{l}} := & x \perp y & x_{\mathbf{r}} \perp y_{\mathbf{r}} := & x \perp y \wedge x \neq \emptyset \wedge y \neq \emptyset \\
x_{\mathbf{l}} \perp y_{\mathbf{r}} := & x = \emptyset \wedge \mathsf{valid}\ y \wedge y \neq \emptyset & x_{\mathbf{r}} \perp y_{\mathbf{l}} := & \mathsf{valid}\ x \wedge x \neq \emptyset \wedge y = \emptyset \\
x_{\mathbf{l}} \cup y_{\mathbf{l}} := & (x \cup y)_{\mathbf{l}} & x_{\mathbf{r}} \cup y_{\mathbf{r}} := & (x \cup y)_{\mathbf{r}} \\
x_{\mathbf{l}} \cup y_{\mathbf{r}} := & y_{\mathbf{r}} & x_{\mathbf{r}} \cup y_{\mathbf{l}} := & x_{\mathbf{r}} \\
\mathsf{splittable}\ x_{\mathbf{l}} := & \mathsf{splittable}\ x & \mathsf{splittable}\ x_{\mathbf{r}} := & \mathsf{splittable}\ x \wedge x = \emptyset \\
\tfrac{1}{2}(x_{\mathbf{l}}) := & (\tfrac{1}{2}x)_{\mathbf{l}} & \tfrac{1}{2}(x_{\mathbf{r}}) := & (\tfrac{1}{2}x)_{\mathbf{r}} \\
\mathsf{unmapped}\ x_{\mathbf{l}} := & \mathsf{unmapped}\ x & \mathsf{unmapped}\ x_{\mathbf{r}} := & \mathsf{unmapped}\ x \wedge x \neq \emptyset \\
\mathsf{exclusive}\ x_{\mathbf{l}} := & \mathsf{exclusive}\ x & \mathsf{exclusive}\ x_{\mathbf{r}} := & \mathsf{exclusive}\ x
\end{array}$$

Definition 4.5.4. *The operations on the CH$_2$O permission system* $\mathsf{kind} : \mathsf{perm} \to \mathsf{pkind}$, $\mathsf{lock}, \mathsf{unlock} : \mathsf{perm} \to \mathsf{perm}$ *and* $\mathsf{token} : \mathsf{perm}$ *are defined as:*

$$\mathsf{kind}\ \gamma := \begin{cases}
\mathsf{Writable} & \text{if } \gamma = \lozenge(x,\, 1) \\
\mathsf{Readable} & \text{if } \gamma = \lozenge(x,\, y) \text{ with } 0 < y < 1 \text{ or } \gamma = x \\
\mathsf{Existing} & \text{if } \gamma = \lozenge(x,\, 0) \text{ with } x \neq 0 \\
\mathsf{Locked} & \text{if } \gamma = \blacklozenge(x,\, y) \\
\mathsf{Readable} & \text{if } \gamma \in \mathbb{Q} \\
\perp & \text{otherwise}
\end{cases}$$

$$\begin{aligned}
\mathsf{lock}\ \gamma &:= \begin{cases} \blacklozenge(x,\, y) & \text{if } \gamma = \lozenge(x,\, y) \\ \gamma & \text{otherwise} \end{cases} \\
\mathsf{unlock}\ \gamma &:= \begin{cases} \lozenge(x,\, y) & \text{if } \gamma = \blacklozenge(x,\, y) \\ \gamma & \text{otherwise} \end{cases} \\
\mathsf{token} &:= \lozenge(-1,\, 0)
\end{aligned}$$

Lemma 4.5.5. *The CH$_2$O permission system satisfies the following properties:*

$$\begin{aligned}
\mathsf{unlock}\ (\mathsf{lock}\ x) &= x & &\text{if } \mathsf{Writable} \subseteq \mathsf{kind}\ x \\
\mathsf{kind}\ (\mathsf{lock}\ x) &= \mathsf{Locked} & &\text{if } \mathsf{Writable} \subseteq \mathsf{kind}\ x \\
\mathsf{kind}\ \left(\tfrac{1}{2}x\right) &= \begin{cases} \mathsf{Readable} & \text{if } \mathsf{Writable} \subseteq \mathsf{kind}\ x \\ \mathsf{kind}\ x & \text{otherwise} \end{cases} \\
\mathsf{kind}\ \mathsf{token} &= \mathsf{Existing} \\
\mathsf{kind}\ (x \setminus \mathsf{token}) &= \mathsf{kind}\ x \\
\mathsf{kind}\ x_1 &\subseteq \mathsf{kind}\ x_2 & &\text{if } x_1 \subseteq x_2
\end{aligned}$$

4.6 Reasoning about disjointness

To prove soundness of our axiomatic semantics in Chapter 8 we often have to reason about preservation of disjointness under memory operations. This section describes some machinery to ease reasoning about disjointness. We show that our machinery, as originally developed in [Kre14a], extends to any separation algebra.

Definition 4.6.1. Disjointness of a list $\vec{x}$, *notation* $\perp \vec{x}$, *is defined as:*

1. $\perp \varepsilon$
2. *If* $\perp \vec{x}$ *and* $x \perp \bigcup \vec{x}$*, then* $\perp (x\, \vec{x})$

Notice that $\perp \vec{x}$ is stronger than having $x_i \perp x_j$ for each $i \neq j$. For example, using fractional permissions, we do not have $\perp [\, 0.5,\, 0.5,\, 0.5\,]$ whereas $0.5 \perp 0.5$ clearly holds. Using disjointness of lists we can for example state the associativity law (law 3 of Definition 4.2.1) in a symmetric way:

Fact 4.6.2. *If* $\perp (x\ y\ z)$*, then* $x \cup (y \cup z) = (x \cup y) \cup z$.

We define a relation $\vec{x}_1 \equiv_\perp \vec{x}_2$ that expresses that $\vec{x}_1$ and $\vec{x}_2$ behave equivalently with respect to disjointness.

Definition 4.6.3. Equivalence of lists $\vec{x}_1$ and $\vec{x}_2$ with respect to disjointness*, notation* $\vec{x}_1 \equiv_\perp \vec{x}_2$*, is defined as:*

$$\begin{aligned}
\vec{x}_1 \leq_\perp \vec{x}_2 &:= \forall x\, .\perp (x\, \vec{x}_1) \to \perp (x\, \vec{x}_2) \\
\vec{x}_1 \equiv_\perp \vec{x}_2 &:= \vec{x}_1 \leq_\perp \vec{x}_2 \wedge \vec{x}_2 \leq_\perp \vec{x}_1
\end{aligned}$$

It is straightforward to show that $\leq_\perp$ is reflexive and transitive, is respected by concatenation of lists, and is preserved by list containment. Hence, $\equiv_\perp$ is an equivalence relation, a congruence with respect to concatenation of lists, and is preserved by permutations. The following results (on arbitrary separation algebras) allow us to reason algebraically about disjointness.

Fact 4.6.4. *If $\vec{x}_1 \leq_\perp \vec{x}_2$, then $\perp \vec{x}_1$ implies $\perp \vec{x}_2$.*

Fact 4.6.5. *If $\vec{x}_1 \equiv_\perp \vec{x}_2$, then $\perp \vec{x}_1$ iff $\perp \vec{x}_2$.*

Theorem 4.6.6. *We have the following algebraic properties:*

$$\begin{array}{rl}
\emptyset \equiv_\perp \varepsilon & \\
x_1 \cup x_2 \equiv_\perp x_1\, x_2 & \text{provided that } x_1 \perp x_2 \\
\bigcup \vec{x} \equiv_\perp \vec{x} & \text{provided that} \perp \vec{x} \\
x_2 \equiv_\perp x_1\, (x_2 \setminus x_1) & \text{provided that } x_1 \subseteq x_2
\end{array}$$

In Section 8.1 we show that we have similar properties as the above for the specific operations of our memory model.

Chapter 5

The memory model

The memory model is the core of a semantics of an imperative programming language. It models the memory states and describes the behavior of memory operations. The main operations described by the CH_2O memory model are:

- Reading a value at a given address.
- Storing a value at a given address.
- Allocating a new object to hold a local variable or storage obtained via `malloc`.
- Deallocating a previously allocated object.

Formalizing the C11 memory model in a faithful way is a challenging task because C features both *low-level* and *high-level* data access. Low-level data access involves unstructured and untyped byte representations whereas high-level data access involves typed abstract values such as arrays, structs and unions.

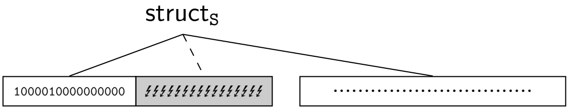
This duality makes the memory model of C more complicated than the memory model of nearly any other programming language. For example, more mathematically oriented languages such as Java and ML feature only high-level data access whereas assembly languages feature only low-level data access.

The situation becomes more complicated as the C11 standard allows compilers to perform optimizations based on a high-level view of data access that are inconsistent with the traditional low-level view of data access. In Chapter 2 we have shown that widely used C compilers actively perform such optimizations. It is therefore essential to faithfully describe the interaction between both views of data access.

In order to formalize the interaction between low-level and high-level data access we represent the formal memory state as a forest of well-typed trees whose structure corresponds to the structure of data types in C. The leaves of these trees consist of bits to capture low-level aspects of the language.

The key concepts of our memory model are as follows.

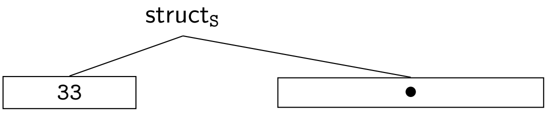
- *Memory trees* (Section 5.4) are used to represent each object in memory. They are abstract trees whose structure corresponds to the shape of C data types. The memory tree of `struct S { short x, *r; } s = { 33, &s.x }` might be (the precise shape and the bit representations are implementation-defined):



The leaves of memory trees contain permission annotated bits (Section 5.3). Bits are represented symbolically: the integer value `33` is represented as its binary representation `1000010000000000`, the padding bytes as symbolic *indeterminate* bits ↯ (whose actual value should not be used), and the pointer `&s.x` as a sequence of symbolic *pointer bits*.

The *memory* itself is a forest of memory trees. Memory trees are explicit about type information (in particular the variants of unions) and thus give rise to a natural formalization of effective types.

- *Pointers* (Section 5.2) are formalized using paths through memory trees. Since we represent pointers as paths, the formal representation contains detailed information about how each pointer has been obtained (in particular which variants of unions were used). A detailed formal representation of pointers is essential to describe effective types.
- *Abstract values* (Definition 5.5) are trees whose structure is similar to memory trees, but have base values (mathematical integers and pointers) on their leaves. The abstract value of `struct S { short x, *r; } s = { 33, &s.x }` is:



Abstract values hide internal details of the memory such as permissions, padding and object representations. They are therefore used in the external interface of the memory model and throughout the operational semantics.

Memory trees, abstract values and bits with permissions can be converted into each other. These conversions are used to define operations internal to the memory model. However, none of these conversions are bijective because different information is materialized in these three data types:

	Abstract values	Memory trees	Bits with permissions
Permissions		✓	✓
Padding		always ↯	✓
Variants of union	✓	✓	
Mathematical values	✓		

This table indicates that abstract values and sequences of bits are complementary. Memory trees are a middle ground, and therefore suitable to describe both the low-level and high-level aspects of the C memory.

5.1 Interface of the memory model

This chapter defines the CH$_2$O memory model whose external interface consists of operations with the following types:

$$\begin{aligned}
\mathsf{lookup}_\Gamma &: \mathsf{addr} \to \mathsf{mem} \to \mathsf{option}\ \mathsf{val}\\
\mathsf{force}_\Gamma &: \mathsf{addr} \to \mathsf{mem} \to \mathsf{mem}\\
\mathsf{insert}_\Gamma &: \mathsf{addr} \to \mathsf{mem} \to \mathsf{val} \to \mathsf{mem}\\
\mathsf{writable}_\Gamma &: \mathsf{addr} \to \mathsf{mem} \to \mathsf{Prop}\\
\mathsf{lock}_\Gamma &: \mathsf{addr} \to \mathsf{mem} \to \mathsf{mem}\\
\mathsf{unlock} &: \mathsf{lockset} \to \mathsf{mem} \to \mathsf{mem}\\
\mathsf{alloc}_\Gamma &: \mathsf{index} \to \mathsf{val} \to \mathsf{bool} \to \mathsf{mem} \to \mathsf{mem}\\
\mathsf{dom} &: \mathsf{mem} \to \mathcal{P}_{\mathsf{fin}}(\mathsf{index})\\
\mathsf{freeable} &: \mathsf{addr} \to \mathsf{mem} \to \mathsf{Prop}\\
\mathsf{free} &: \mathsf{index} \to \mathsf{mem} \to \mathsf{mem}
\end{aligned}$$

Notation 5.1.1. *We let $m\langle a\rangle_\Gamma := \mathsf{lookup}_\Gamma\, a\, m$ and $m\langle a := v\rangle_\Gamma := \mathsf{insert}_\Gamma\, a\, v\, m$.*

Many of these operations depend on the typing environment Γ which assigns fields to structs and unions (Definition 3.3.3 on page 45). This dependency is required as these operations need to be aware of the layout of structs and unions.

The operation $m\langle a\rangle_\Gamma$ yields the value stored at address a in memory m. It fails with $\bot$ if the permissions are insufficient, effective types are violated, or a is an end-of-array address. Reading from (the abstract) memory is not a pure operation. Although it does not affect the memory contents, it may affect the effective types [ISO12, 6.5p6-7]. This happens for example in case type-punning is performed (see Section 2.5.6). This impurity is factored out by the operation $\mathsf{force}_\Gamma\, a\, m$.

The operation $m\langle a := v\rangle_\Gamma$ stores the value v at address a in memory m. A store is only permitted in case permissions are sufficient, effective types are not violated, and a is not an end-of-array address. The proposition $\mathsf{writable}_\Gamma\, a\, m$ describes the side-conditions necessary to perform a store.

After a successful store, the operation $\mathsf{lock}_\Gamma\, a\, m$ is used to lock the object at address a in memory m. The lock operation temporarily reduces the permissions to Locked so as to prohibit future accesses to a. Locking yields a formal treatment of the sequence point restriction (which states that modifying an object more than once between two sequence points results in undefined behavior, see Section 2.5.9).

The operational semantics accumulates a set $\Omega \in \mathsf{lockset}$ of addresses that have been written to (Definition 5.6.1) and uses the operation $\mathsf{unlock}\ \Omega\ m$ at the subsequent sequence point (which may be at the semicolon that terminates a full expression). The operation $\mathsf{unlock}\ \Omega\ m$ restores the permissions of the addresses in Ω and thereby makes future accesses to the addresses in Ω possible again. In Chapter 6, we describe how we treat sequence points and locks in the operational semantics.

The operation $\mathsf{alloc}_\Gamma\, o\ v\ \mu\ m$ allocates a new object with value v in memory m. The object has object identifier $o \notin \mathsf{dom}\ m$ which is non-deterministically chosen by the operation semantics. The Boolean μ expresses if the new object is allocated by `malloc` (the expression $\mathsf{alloc}_\tau\ e$ in our language, see Definition 6.1.4 on page 95).

Accompanying alloc_Γ, the operation $\mathsf{free}\ o\ m$ deallocates a previously allocated object with object identifier o in memory m. In order to deallocate dynamically obtained memory via `free`, the side-condition $\mathsf{freeable}\ a\ m$ describes that the permissions are sufficient for deallocation, and that a points to a `malloc`ed object.

5.2 Representation of pointers

Adapted from CompCert [LB08, LABS12], we represent memory states as finite partial functions from *object identifiers* to *objects*. Each local, global and static variable, as well as each invocation of `malloc`, is associated with a unique object identifier of a separate object in memory. This approach separates unrelated objects by construction, and is therefore well-suited for reasoning about memory transformations.

We improve on CompCert by modeling objects as structured trees instead of arrays of bytes to keep track of padding bytes and the variants of unions. This is needed to faithfully describe C11's notion of effective types (see Section 2.5.5 for an informal description). This approach allows us to describe various undefined behaviors of C11 that have not been considered by others (see Sections 2.5.2 and 2.5.6).

In the CompCert memory model, pointers are represented as pairs (o, i) where o is an object identifier and i is a byte offset into the object with object identifier o. Since we represent objects as trees instead of as arrays of bytes, we represent pointers as paths through these trees rather than as byte offsets.

Definition 5.2.1. Object identifiers $o \in \mathsf{index}$ *are elements of a fixed countable set. In the Coq development we use binary natural numbers, but since we do not rely on any properties apart from countability, we keep the representation opaque.*

We first introduce a typing environment to relate the shape of paths representing pointers to the types of objects in memory.

Definition 5.2.2. Memory typing environments $\Delta \in \mathsf{memenv}$ *are finite partial functions* $\mathsf{index} \rightarrow_{\mathsf{fin}} (\mathsf{type} \times \mathsf{bool})$. *Given a memory environment* Δ*:*

1. *An* object identifier o has type τ, *notation* $\Delta \vdash o : \tau$, *if* $\Delta\, o = (\tau, \beta)$ *for a* β.
2. *An* object identifier o is alive, *notation* $\Delta \vdash o\ \mathsf{alive}$, *if* $\Delta\, o = (\tau, \mathsf{false})$ *for a* τ.

Memory typing environments evolve during program execution. The code below is annotated with the corresponding memory environments in red.

```
short x;
```

$$\Delta_1 = \{o_1 \mapsto (\mathsf{signed\ short}, \mathsf{false})\}$$

```
int *p;
```

$$\Delta_2 = \{o_1 \mapsto (\mathsf{signed\ short}, \mathsf{false}), o_2 \mapsto (\mathsf{signed\ int}{*}, \mathsf{false})\}$$

```
p = malloc(sizeof(int));
```

$$\Delta_3 = \{o_1 \mapsto (\mathsf{signed\ short}, \mathsf{false}), o_2 \mapsto (\mathsf{signed\ int}{*}, \mathsf{false}), o_3 \mapsto (\mathsf{signed\ int}, \mathsf{false})\}$$

```
free(p);
```

$$\Delta_4 = \{o_1 \mapsto (\mathsf{signed\ short}, \mathsf{false}), o_2 \mapsto (\mathsf{signed\ int}{*}, \mathsf{false}), o_3 \mapsto (\mathsf{signed\ int}, \mathsf{true})\}$$

Here, o_1 is the object identifier of the variable `x`, o_2 is the object identifier of the variable `p` and o_3 is the object identifier of the storage obtained via `malloc`.

Memory typing environments also keep track of objects that have been deallocated. Although one cannot directly create a pointer to a deallocated object, existing pointers to such objects remain in memory after deallocation (see the pointer `p` in the above example). These pointers, also called *dangling* pointers, cannot actually be used, see the discussion in Section 2.4.2.

Definition 5.2.3. References, addresses *and* pointers *are inductively defined as:*

$$
\begin{aligned}
r \in \mathsf{refseg} &::= \xhookrightarrow{\tau[n]} i \mid \xhookrightarrow{\mathsf{struct}\ t} i \mid \xhookrightarrow{\mathsf{union}\ t}_q i \text{ with } q \in \{\circ, \bullet\} \\
\vec{r} \in \mathsf{ref} &:= \mathsf{list\ refseg} \\
a \in \mathsf{addr} &::= (o : \tau, \vec{r}, i)_{\sigma >_* \sigma_{\mathsf{p}}} \\
p \in \mathsf{ptr} &::= \mathsf{NULL}\ \sigma_{\mathsf{p}} \mid a \mid f^{\vec{\tau} \mapsto \tau}
\end{aligned}
$$

References are paths from the top of an object in memory to some subtree of that object. The shape of references matches the structure of types:

- The reference $\xhookrightarrow{\tau[n]} i$ is used to select the ith element of a τ-array of length n.
- The reference $\xhookrightarrow{\mathsf{struct}\ t} i$ is used to select the ith field of a struct t.
- The reference $\xhookrightarrow{\mathsf{union}\ t}_q i$ is used to select the ith variant of a union t.

References can describe most pointers in C but cannot account for end-of-array pointers and pointers to individual bytes. We have therefore defined the richer notion of *addresses*. An address $(o : \tau, \vec{r}, i)_{\sigma >_* \sigma_{\mathsf{p}}}$ consists of:

- An object identifier o with type τ.
- A reference $\vec{r}$ to a subobject of type σ in the entire object of type τ.
- An offset i to a particular byte in the subobject of type σ (note that one cannot address individual bits in C).
- The type σ_{p} to which the address is cast. We use a points-to type in order to account for casts to the anonymous **`void*`** pointer, which is represented as the points-to type any. This information is needed to define, for example, pointer arithmetic, which is sensitive to the type of the address.

In turn, pointers extend addresses with a `NULL` pointer $\mathsf{NULL}\ \sigma_{\mathsf{p}}$ for each type σ_{p}, and function pointers $f^{\vec{\tau} \mapsto \tau}$ which contain the name and type of a function.

Let us consider the following global variable declaration:

```
struct S {
  union U { signed char x[2]; int y; } u;
  void *p;
} s;
```

The formal representation of the pointer `(void*)(s.u.x + 2)` is:

$$(o_{\mathsf{s}} : \mathsf{struct\ S}, \xhookrightarrow{\mathsf{struct\ S}} 0 \xhookrightarrow{\mathsf{union\ U}}_{\bullet} 0 \xhookrightarrow{\mathsf{signed\ char}[2]} 0, 2)_{\mathsf{signed\ char} >_* \mathsf{any}}.$$

Here, $o_{\mathtt{s}}$ is the object identifier associated with the variable `s` of type $\mathsf{struct}\ \mathtt{S}$. The reference $\xhookrightarrow{\mathsf{struct}\ \mathtt{S}} 0 \xhookrightarrow{\mathsf{union}\ \mathtt{U}}_{\bullet} 0 \xhookrightarrow{\mathsf{signed\ char}[2]} 0$ and byte-offset 2 describe that the pointer refers to the third byte of the array `s.u.x`. The pointer refers to an object of type $\mathsf{signed\ char}$. The annotation any describes that the pointer has been cast to **`void*`**.

The annotations $q \in \{\circ, \bullet\}$ on references $\xhookrightarrow{\mathsf{union}\ s}_{q} i$ describe whether type-punning is allowed or not. The annotation $\bullet$ means that type-punning is allowed, *i.e.* accessing another variant than the current one has defined behavior. The annotation $\circ$ means that type-punning is forbidden. A pointer whose annotations are all of the shape $\circ$, and thereby does not allow type-punning at all, is called *frozen*.

Definition 5.2.4. *The* freeze *function* $|\,\_\,|_{\circ} : \mathsf{refseg} \to \mathsf{refseg}$ *is defined as:*

$$|\xhookrightarrow{\tau[n]} i\,|_{\circ} := \xhookrightarrow{\tau[n]} i \qquad |\xhookrightarrow{\mathsf{struct}\ t} i\,|_{\circ} := \xhookrightarrow{\mathsf{struct}\ t} i \qquad |\xhookrightarrow{\mathsf{union}\ t}_{q} i\,|_{\circ} := \xhookrightarrow{\mathsf{union}\ t}_{\circ} i$$

A reference segment r is frozen*, notation* $\mathsf{frozen}\ r$*, if* $|\,r\,|_{\circ} = r$*. Both* $|\,\_\,|_{\circ}$ *and* frozen *are lifted to references, addresses, and pointers in the expected way.*

Pointers stored in memory are always in frozen shape. Definitions 5.4.10 and 5.5.3 describe the formal treatment of effective types and frozen pointers, but for now we reconsider the example from Section 2.5.6:

```
union U { int x; short y; } u = { .x = 3 };
short *p = &u.y;
printf("%d\n", *p);  // Undefined
printf("%d\n", u.y); // OK
```

Assuming the object `u` has object identifier $o_{\mathtt{u}}$, the pointers `&u.x`, `&u.y` and `p` have the following formal representations:

$$\begin{aligned}
\texttt{\&u.x:}\quad & (o_{\mathtt{u}} : \mathsf{union}\ \mathtt{U}, \xhookrightarrow{\mathsf{union}\ \mathtt{U}}_{\bullet} 0, 0)_{\mathsf{signed\ int} >_{*} \mathsf{signed\ int}}\\
\texttt{\&u.y:}\quad & (o_{\mathtt{u}} : \mathsf{union}\ \mathtt{U}, \xhookrightarrow{\mathsf{union}\ \mathtt{U}}_{\bullet} 1, 0)_{\mathsf{signed\ short} >_{*} \mathsf{signed\ short}}\\
\texttt{p:}\quad & (o_{\mathtt{u}} : \mathsf{union}\ \mathtt{U}, \xhookrightarrow{\mathsf{union}\ \mathtt{U}}_{\circ} 1, 0)_{\mathsf{signed\ short} >_{*} \mathsf{signed\ short}}
\end{aligned}$$

These pointers are likely to have the same object representation on actual computing architectures. However, due to effective types, `&u.y` may be used for type-punning but `p` may not. It is thus important that we distinguish these pointers in the formal memory model.

The additional structure of pointers is also needed to determine whether pointer subtraction has defined behavior. The behavior is only defined if the given pointers both point to an element of the same array object [ISO12, 6.5.6p9]. Consider:

```
struct S { int a[3]; int b[3]; } s;
s.a - s.b;       // Undefined, different array objects
(s.a + 3) - s.b; // Undefined, different array objects
(s.a + 3) - s.a; // OK, same array objects
```

Here, the pointers `s.a + 3` and `s.b` have different representations in the memory model. Definition 6.3.2 on page 102 gives the formal definition of pointer subtraction.

We will now define typing judgments for references, addresses and pointers. The judgment for references $\Gamma \vdash \vec{r} : \tau \rightarrowtail \sigma$ states that σ is a *subobject type of* τ which can be obtained via the reference $\vec{r}$ (see also Definition 5.7.1). For example, **`int`**`[2]` is a subobject type of **`struct`** `S {` **`int`** `x[2];` **`int`** `y[3]; }` via $\xhookrightarrow{\mathsf{struct}\ \mathsf{S}} 0$.

Definition 5.2.5. *The judgment* $\Gamma \vdash \vec{r} : \tau \rightarrowtail \sigma$ *describes that* $\vec{r}$ is a valid reference from τ to σ. *It is inductively defined as:*

$$\frac{}{\Gamma \vdash \varepsilon : \tau \rightarrowtail \tau} \qquad \frac{\Gamma \vdash \vec{r} : \tau \rightarrowtail \sigma[n] \qquad i < n}{\Gamma \vdash \vec{r} \xhookrightarrow{\sigma[n]} i : \tau \rightarrowtail \sigma}$$

$$\frac{\Gamma \vdash \vec{r} : \tau \rightarrowtail \mathsf{struct}\ t \quad \Gamma\, t = \vec{\sigma} \quad i < |\vec{\sigma}|}{\Gamma \vdash \vec{r} \xhookrightarrow{\mathsf{struct}\ t} i : \tau \rightarrowtail \sigma_i} \qquad \frac{\Gamma \vdash \vec{r} : \tau \rightarrowtail \mathsf{union}\ t \quad \Gamma\, t = \vec{\sigma} \quad i < |\vec{\sigma}|}{\Gamma \vdash \vec{r} \xhookrightarrow{\mathsf{union}\ t}_q i : \tau \rightarrowtail \sigma_i}$$

The typing judgment for addresses is more involved than the judgment for references. Let us first consider the following example:

```
int a[4];
```

Assuming the object `a` has object identifier o_{a}, the end-of-array pointer `a+4` could be represented in at least the following ways (assuming $\mathsf{sizeof}\ (\mathsf{signed\ int}) = 4$):

$$(o_{\mathsf{a}} : \mathsf{signed\ int}[4], \xhookrightarrow{\mathsf{signed\ int}[4]} 0, 16)_{\mathsf{signed\ int} >_* \mathsf{signed\ int}}$$
$$(o_{\mathsf{a}} : \mathsf{signed\ int}[4], \xhookrightarrow{\mathsf{signed\ int}[4]} 3, 4)_{\mathsf{signed\ int} >_* \mathsf{signed\ int}}$$

In order to ensure canonicity of pointer representations, we let the typing judgment for addresses ensure that the reference $\vec{r}$ of $(o : \tau, \vec{r}, i)_{\sigma >_* \sigma_{\mathsf{p}}}$ always refers to the first element of an array subobject. This renders the second representation illegal.

Definition 5.2.6. *The relation* $\tau >_* \sigma_{\mathsf{p}}$*, type* τ is pointer castable to σ_{p}*, is inductively defined by* $\tau >_* \tau$*,* $\tau >_* \mathsf{unsigned\ char}$*, and* $\tau >_* \mathsf{any}$*.*

Definition 5.2.7. *The judgment* $\Gamma, \Delta \vdash_* a : \sigma_{\mathsf{p}}$ *describes that* the address a refers to type σ_{p}. *It is inductively defined as:*

$$\frac{\begin{array}{c}\Delta \vdash o : \tau \qquad \Gamma \vdash \tau \qquad \Gamma \vdash \vec{r} : \tau \rightarrowtail \sigma \\ \mathsf{offset}\ \vec{r} = 0 \qquad i \leq \mathsf{sizeof}_\Gamma\ \sigma \cdot \mathsf{size}\ \vec{r} \qquad \mathsf{sizeof}_\Gamma\ \sigma_{\mathsf{p}} \mid i \qquad \sigma >_* \sigma_{\mathsf{p}}\end{array}}{\Gamma, \Delta \vdash (o : \tau, \vec{r}, i)_{\sigma >_* \sigma_{\mathsf{p}}} : \sigma_{\mathsf{p}}}$$

Here, the helper functions $\mathsf{offset}, \mathsf{size} : \mathsf{ref} \to \mathbb{N}$ *are defined as:*

$$\mathsf{offset}\ \vec{r} := \begin{cases} i & \text{if } \vec{r} = \vec{r}_2 \xhookrightarrow{\tau[n]} i \\ 0 & \text{otherwise} \end{cases} \qquad \mathsf{size}\ \vec{r} := \begin{cases} n & \text{if } \vec{r} = \vec{r}_2 \xhookrightarrow{\tau[n]} i \\ 1 & \text{otherwise} \end{cases}$$

We use an intrinsic encoding of syntax, which means that terms contain redundant type annotations so we can read off types. Functions to read off types are named typeof and will not be defined explicitly. Type annotations make it more convenient to define operations that depend on types (such as offset and size in Definition 5.2.7). As usual, typing judgments ensure that type annotations are consistent.

The premises $i \leq \mathsf{sizeof}_\Gamma\ \sigma \cdot \mathsf{size}\ \vec{r}$ and $\mathsf{sizeof}_\Gamma\ \sigma_\mathsf{p} \mid i$ of the typing rule ensure that the byte offset i is aligned and within range. The inequality $i \leq \mathsf{sizeof}_\Gamma\ \sigma \cdot \mathsf{size}\ \vec{r}$ is non-strict so as to allow end-of-array pointers.

Definition 5.2.8. *An address* $a = (o : \tau, \vec{r}, i)_{\sigma >_* \sigma_\mathsf{p}}$ *is called* strict*, notation* $\Gamma \vdash a\ \mathsf{strict}$*, in case it satisfies* $i < \mathsf{sizeof}_\Gamma\ \sigma \cdot \mathsf{size}\ \vec{r}$*.*

The judgment $\tau >_* \sigma_\mathsf{p}$ does not describe the typing restriction of cast expressions. Instead, it defines the invariant that each address $(o : \tau, \vec{r}, i)_{\sigma >_* \sigma_\mathsf{p}}$ should satisfy. Since C is not type safe, the operation for pointer casting (Definition 6.3.4 on page 103) has $\tau >_* \sigma_\mathsf{p}$ as a side-condition:

```
int x, *p = &x;
void *q = (void*)p;     // OK, signed int >* any
int *q1 = (int*)q;      // OK, signed int >* signed int
short *q2 = (short*)p; // Statically ill-typed
short *q3 = (short*)q; // Undefined behavior, signed int ≯* signed short
```

Definition 5.2.9. *The judgment* $\Gamma, \Delta \vdash_* p : \sigma_\mathsf{p}$ *describes that* the pointer p refers to type σ_p*. It is inductively defined as:*

$$\frac{\Gamma \vdash_* \sigma_\mathsf{p}}{\Gamma, \Delta \vdash_* \mathsf{NULL}\ \sigma_\mathsf{p} : \sigma_\mathsf{p}} \qquad \frac{\Gamma, \Delta \vdash a : \sigma_\mathsf{p}}{\Gamma, \Delta \vdash_* a : \sigma_\mathsf{p}} \qquad \frac{\Gamma\, f = (\vec{\tau}, \tau)}{\Gamma, \Delta \vdash_* f^{\vec{\tau} \mapsto \tau} : \vec{\tau} \to \tau}$$

Addresses $(o : \tau, \vec{r} \xhookrightarrow{\sigma[n]} j, i)_{\sigma >_* \sigma_\mathsf{p}}$ that point to an element of $\tau[n]$ always have their reference point to the first element of the array, *i.e.* $j = 0$. For some operations we use the *normalized reference* which refers to the actual array element.

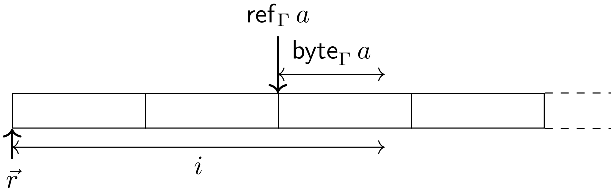
Definition 5.2.10. *The functions* $\mathsf{index} : \mathsf{addr} \to \mathsf{index}$*,* $\mathsf{ref}_\Gamma : \mathsf{addr} \to \mathsf{ref}$*, and* $\mathsf{byte}_\Gamma : \mathsf{addr} \to \mathbb{N}$ *obtain the* index*,* normalized reference*, and* normalized byte offset*.*

$$\begin{aligned} \mathsf{index}\ (o : \tau, \vec{r}, i)_{\sigma >_* \sigma_\mathsf{p}} &:= o \\ \mathsf{ref}_\Gamma\ (o : \tau, \vec{r}, i)_{\sigma >_* \sigma_\mathsf{p}} &:= \mathsf{setoffset}\ (i \div \mathsf{sizeof}_\Gamma\ \sigma)\ \vec{r} \\ \mathsf{byte}_\Gamma\ (o : \tau, \vec{r}, i)_{\sigma >_* \sigma_\mathsf{p}} &:= i\ \mathsf{mod}\ (\mathsf{sizeof}_\Gamma\ \sigma) \end{aligned}$$

Here, the function $\mathsf{setoffset} : \mathbb{N} \to \mathsf{ref} \to \mathsf{ref}$ *is defined as:*

$$\mathsf{setoffset}\, j\ \vec{r} := \begin{cases} \vec{r_2} \xhookrightarrow{\tau[n]} j & \text{if } \vec{r} = \vec{r_2} \xhookrightarrow{\tau[n]} i \\ r & \text{otherwise} \end{cases}$$

Let us display the above definition graphically. Given an address $(o : \tau, \vec{r}, i)_{\sigma >_* \sigma_p}$, the normalized reference and normalized byte offset are as follows:



For end-of-array addresses the normalized reference is ill-typed because references cannot be end-of-array. For strict addresses the normalized reference is well-typed.

Definition 5.2.11. *The judgment* $\Delta \vdash p$ alive *describes that* the pointer p is alive. *It is inductively defined as:*

$$\frac{}{\Delta \vdash \mathsf{NULL}\ \sigma_p\ \mathsf{alive}} \qquad \frac{\Delta \vdash (\mathsf{index}\ a)\ \mathsf{alive}}{\Delta \vdash a\ \mathsf{alive}} \qquad \frac{}{\Delta \vdash f^{\vec{\tau} \mapsto \tau}\ \mathsf{alive}}$$

The judgment $\Delta \vdash o$ alive *on object identifiers is defined in Definition 5.2.2.*

For many operations we have to distinguish addresses that refer to an entire object and addresses that refer to an individual byte of an object. We call addresses of the later kind *byte addresses*. For example:

```
int x, *p = &x;                         // p is not a byte address
unsigned char *q = (unsigned char*)&x; // q is a byte address
```

Definition 5.2.12. *An address* $(o : \tau, \vec{r}, i)_{\sigma >_* \sigma_p}$ *is* a byte address *if* $\sigma \neq \sigma_p$.

To express that memory operations commute (see for example Lemma 5.4.14), we need to express that addresses are *disjoint*, meaning they do not overlap. Addresses do not overlap if they belong to different objects or take a different branch at an array or struct. Let us consider an example:

```
union { struct { int x, y; } s; int z; } u1, u2;
```

The pointers `&u1` and `&u2` are disjoint because they point to separate memory objects. Writing to one does not affect the value of the other and *vice versa*. Likewise, `&u1.s.x` and `&u1.s.y` are disjoint because they point to different fields of the same struct, and as such do not affect each other. The pointers `&u1.s.x` and `&u1.z` are not disjoint because they point to overlapping objects and thus do affect each other.

Definition 5.2.13. Disjointness of references $\vec{r}_1$ and $\vec{r}_2$, *notation* $\vec{r}_1 \perp \vec{r}_2$, *is inductively defined as:*

$$\frac{|\,\vec{r}_1\,|_\circ = |\,\vec{r}_2\,|_\circ \qquad i \neq j}{\vec{r}_1 \xhookrightarrow{\sigma[n]} i\,\vec{r}_3 \perp \vec{r}_2 \xhookrightarrow{\sigma[n]} j\,\vec{r}_4} \qquad \frac{|\,\vec{r}_1\,|_\circ = |\,\vec{r}_2\,|_\circ \qquad i \neq j}{\vec{r}_1 \xhookrightarrow{\mathsf{struct}\ t} i\,\vec{r}_3 \perp \vec{r}_2 \xhookrightarrow{\mathsf{struct}\ t} j\,\vec{r}_4}$$

Note that we do not require a special case for $|\,\vec{r_1}\,|_\circ \neq |\,\vec{r_2}\,|_\circ$. Such a case is implicit because disjointness is defined in terms of prefixes.

Definition 5.2.14. Disjointness of addresses a_1 and a_2, *notation* $a_1 \perp_\Gamma a_2$, *is inductively defined as:*

$$\frac{\mathsf{index}\ a_1 \neq \mathsf{index}\ a_2}{a_1 \perp_\Gamma a_2} \qquad \frac{\mathsf{index}\ a_1 = \mathsf{index}\ a_2 \qquad \mathsf{ref}_\Gamma\ a_1 \perp \mathsf{ref}_\Gamma\ a_2}{a_1 \perp_\Gamma a_2}$$

$$\frac{\begin{array}{c}\textit{both } a_1 \textit{ and } a_2 \textit{ are byte addresses}\\ \mathsf{index}\ a_1 = \mathsf{index}\ a_2 \qquad |\,\mathsf{ref}_\Gamma\ a_1\,|_\circ = |\,\mathsf{ref}_\Gamma\ a_2\,|_\circ \qquad \mathsf{byte}_\Gamma\ a_1 \neq \mathsf{byte}_\Gamma\ a_2\end{array}}{a_1 \perp_\Gamma a_2}$$

The first inference rule accounts for addresses whose object identifiers are different, the second rule accounts for addresses whose references are disjoint, and the third rule accounts for addresses that point to different bytes of the same subobject.

Definition 5.2.15. *The* reference bit-offset $\mathsf{bitoffset}_\Gamma : \mathsf{refseg} \to \mathbb{N}$ *is defined as:*

$$\begin{aligned}
\mathsf{bitoffset}_\Gamma\ (\xhookrightarrow{\tau[n]} i) &:= i \cdot \mathsf{bitsizeof}_\Gamma\ \tau\\
\mathsf{bitoffset}_\Gamma\ (\xhookrightarrow{\mathsf{union}\ t}_q i) &:= 0\\
\mathsf{bitoffset}_\Gamma\ (\xhookrightarrow{\mathsf{struct}\ t} i) &:= \mathsf{bitoffsetof}_\Gamma\ \vec{\tau}\ i \quad \text{where } \Gamma\, t = \vec{\tau}
\end{aligned}$$

Moreover, we let $\mathsf{bitoffset}_\Gamma\ a := \Sigma_i\ (\mathsf{bitoffset}_\Gamma\ (\mathsf{ref}_\Gamma\ a)_i) + \mathsf{byte}_\Gamma\ a \cdot \mathsf{char\_bits}$.

Disjointness implies non-overlapping bit-offsets, but the reverse implication does not always hold because references to different variants of unions are not disjoint. For example, given the declaration `union { struct { int x, y; } s; int z; } u`, the pointers corresponding to `&u.s.y` and `&u.z` are not disjoint.

Lemma 5.2.16. *If* $\Gamma, \Delta \vdash a_1 : \sigma_1$, $\Gamma, \Delta \vdash a_2 : \sigma_2$, $\Gamma \vdash \{a_1, a_2\}\ \mathsf{strict}$, $a_1 \perp_\Gamma a_2$, *and* $\mathsf{index}\ a_1 \neq \mathsf{index}\ a_2$, *then either:*

1. $\mathsf{bitoffset}_\Gamma\ a_1 + \mathsf{bitsizeof}_\Gamma\ \sigma_1 \leq \mathsf{bitoffset}_\Gamma\ a_2$, *or*
2. $\mathsf{bitoffset}_\Gamma\ a_2 + \mathsf{bitsizeof}_\Gamma\ \sigma_2 \leq \mathsf{bitoffset}_\Gamma\ a_1$.

Lemma 5.2.17 (Well-typed addresses are properly aligned)**.** *If* $\Gamma, \Delta \vdash a : \sigma$, *then* $(\mathsf{alignof}_\Gamma\ \sigma \cdot \mathsf{char\_bits}) \mid \mathsf{bitoffset}_\Gamma\ a$.

5.3 Representation of bits

As shown in Section 2.5.2, each object in C can be interpreted as an `unsigned char` array called the *object representation.* On actual computing architectures, the object representation consists of a sequence of concrete bits (zeros and ones). However, so as to accurately describe all undefined behaviors, we need a special treatment for the object representations of pointers and indeterminate memory in the formal memory model. To that end, CH_2O represents the bits belonging to the object representations of pointers and indeterminate memory symbolically.

Definition 5.3.1. Bits *are inductively defined as:*

$$b \in \mathsf{bit} ::= \lightning \mid 0 \mid 1 \mid (\mathsf{ptr}\ p)_i.$$

Definition 5.3.2. *The judgment* $\Gamma, \Delta \vdash b$ *describes that a bit* b *is* valid. *It is inductively defined as:*

$$\frac{}{\Gamma, \Delta \vdash \lightning} \qquad \frac{\beta \in \{0, 1\}}{\Gamma, \Delta \vdash \beta} \qquad \frac{\Gamma, \Delta \vdash_* p : \sigma_{\mathsf{p}} \qquad \mathsf{frozen}\ p \qquad i < \mathsf{bitsizeof}_\Gamma\,(\sigma_{\mathsf{p}}*)}{\Gamma, \Delta \vdash (\mathsf{ptr}\ p)_i}$$

A bit is either a concrete bit 0 or 1, the ith fragment bit $(\mathsf{ptr}\ p)_i$ of a pointer p, or the indeterminate bit $\lightning$. Integers are represented using concrete sequences of bits, and pointers as sequences of fragment bits. Assuming $\mathsf{bitsizeof}\,(\mathsf{signed\ int}*) = 32$, a pointer $p : \mathsf{signed\ int}$ will be represented as the bit sequence $(\mathsf{ptr}\ p)_0 \dots (\mathsf{ptr}\ p)_{31}$, and assuming $\mathsf{bitsizeof}\,(\mathsf{signed\ int}) = 32$ on a little-endian architecture, the integer $33 : \mathsf{signed\ int}$ will be represented as the bit sequence $\mathtt{1000010000000000}$.

The approach using a combination of symbolic and concrete bits is similar to Leroy *et al.* [LABS12] and has the following advantages:

- Symbolic bit representations for pointers avoid the need to clutter the memory model with subtle, implementation-defined, and run-time dependent operations to decode and encode pointers as concrete bit sequences.
- We can precisely keep track of memory areas that are uninitialized. Since these memory areas consist of arbitrary concrete bit sequences on actual machines, most operations on them have undefined behavior.
- While reasoning about program transformations one has to relate the memory states during the execution of the source program to those during the execution of the target program. Program transformations can, among other things, make more memory defined (that is, transform some indeterminate $\lightning$ bits into determinate bits) and relabel the memory. Symbolic bit representations make it easy to deal with such transformations (see Section 5.8).
- It vastly decreases the amount of non-determinism, making it possible to evaluate the memory model as part of an executable semantics (see Section 6.8).
- The use of concrete bit representations for integers still gives a semantics to many low-level operations on integer representations.

A small difference with Leroy *et al.* [LABS12] is that the granularity of our memory model is on the level of bits rather than bytes. Currently we do not make explicit use of this granularity, but it allows us to support bit-fields more faithfully with respect to the C11 standard in future work.

Objects in our memory model are annotated with permissions. We use permission annotations on the level of individual bits, rather than on the level of bytes or entire objects, to obtain the most precise way of permission handling.

Definition 5.3.3. Permission annotated bits *are defined as:*

$$\mathbf{b} \in \mathsf{pbit} := \mathcal{T}^{\lightning}_{\mathsf{bit}}(\mathsf{perm}) = \mathsf{perm} \times \mathsf{bit}.$$

In the above definition, $\mathcal{T}$ is the tagged separation algebra that has been defined in Definition 4.4.3 on page 57. We have spelled out its definition for brevity's sake.

Definition 5.3.4. *The judgment* $\Gamma, \Delta \vdash \mathbf{b}$ *describes that a permission annotated bit* $\mathbf{b}$ *is* valid. *It is inductively defined as:*

$$\frac{\Gamma, \Delta \vdash b \qquad \mathsf{valid}\ \gamma \qquad b = \lightning \text{ in case } \mathsf{unmapped}\ \gamma}{\Gamma, \Delta \vdash (\gamma,\, b)}$$

5.4 Representation of the memory

Memory trees are abstract trees whose structure corresponds to the shape of data types in C. They are used to describe individual objects (base values, arrays, structs, and unions) in memory. The memory is a forest of memory trees.

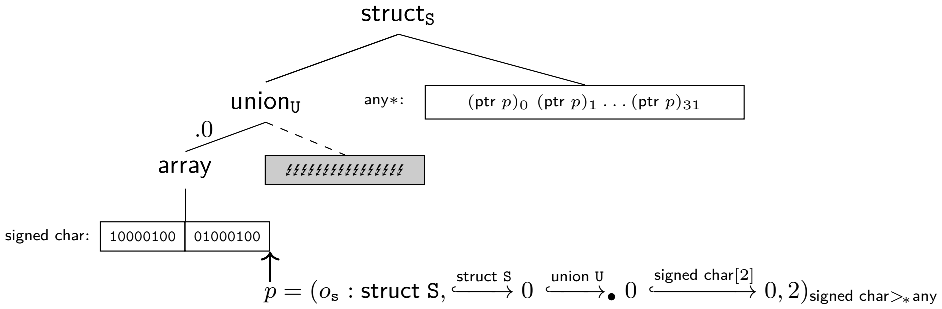
Definition 5.4.1. Memory trees *are inductively defined as:*

$$w \in \mathsf{mtree} ::= \mathsf{base}_{\tau_\mathrm{b}}\ \vec{\mathbf{b}} \mid \mathsf{array}_\tau\ \vec{w} \mid \mathsf{struct}_t\ \overrightarrow{w\mathbf{b}} \mid \mathsf{union}_t\ (i, w, \vec{\mathbf{b}}) \mid \overline{\mathsf{union}_t}\ \vec{\mathbf{b}}.$$

The structure of memory trees is close to the structure of types (Definition 3.3.2 on page 44) and thus reflects the expected semantics of types: arrays are lists, structs are tuples, and unions are sums. Let us consider the following example:

```
struct S {
  union U { signed char x[2]; int y; } u; void *p;
} s = { .u = { .x = {33,34} }, .p = s.u.x + 2 };
```

The memory tree representing the object `s` with object identifier $o_\mathtt{s}$ may be as follows (permissions are omitted for brevity's sake, and integer encoding and padding are subject to implementation-defined behavior):



$\mathsf{struct}_\mathtt{S}$
$\mathsf{union}_\mathtt{U}$
any∗:
$(\mathsf{ptr}\ p)_0\ (\mathsf{ptr}\ p)_1 \ldots (\mathsf{ptr}\ p)_{31}$
.0
array
$\lightning\lightning\lightning\lightning\lightning\lightning\lightning\lightning\lightning\lightning\lightning\lightning\lightning\lightning\lightning\lightning$
signed char:
10000100
01000100
$p = (o_\mathtt{s} : \mathsf{struct}\ \mathtt{S}, \xhookrightarrow{\mathsf{struct}\ \mathtt{S}} 0 \xhookrightarrow{\mathsf{union}\ \mathtt{U}}_\bullet 0 \xhookrightarrow{\mathsf{signed\ char}[2]} 0, 2)_{\mathsf{signed\ char} >_* \mathsf{any}}$

The representation of unions requires some explanation. We considered two kinds of memory trees for unions:

- The memory tree $\mathsf{union}_t\ (i, w, \vec{\mathbf{b}})$ represents a union whose variant is i. Unions of variant i can only be accessed through a pointer to variant i. This is essential for effective types. The list $\vec{\mathbf{b}}$ represents the padding after the element w.

- The memory tree $\overline{\mathsf{union}}_t\,\vec{\mathbf{b}}$ represents a union whose variant is yet unspecified. Whenever the union is accessed through a pointer to variant i, the list $\vec{\mathbf{b}}$ will be interpreted as a memory tree of the type belonging to the ith variant.

The reason that we consider unions $\overline{\mathsf{union}}_t\,\vec{\mathbf{b}}$ with unspecific variant at all is that in some cases the variant cannot be known. Unions that have not been initialized do not have a variant yet. Also, when a union object is constructed byte-wise through its object representation, the variant cannot be known.

Although unions are tagged in the formal memory, actual compilers implement untagged unions. Information about variants should thus be internal to the formal memory model. In Section 5.8 we prove that this is indeed the case.

The additional structure of memory trees, namely type annotations, variants of unions, and structured information about padding, can be erased by flattening. Flattening just appends the bytes on the leaves of the tree.

Definition 5.4.2. *The* flatten operation $\overline{(\_)} : \mathsf{mtree} \to \mathsf{list\ pbit}$ *is defined as:*

$$\overline{\mathsf{base}_{\tau_\mathsf{b}}\,\vec{\mathbf{b}}} := \vec{\mathbf{b}} \qquad \overline{\mathsf{array}_\tau\,\vec{w}} := \overline{w_0}\ldots\overline{w_{|\vec{w}|-1}}$$

$$\overline{\mathsf{struct}_t\,\overrightarrow{w\mathbf{b}}} := (\overline{w_0}\,\vec{\mathbf{b}}_0)\ldots(\overline{w_{|\vec{w}|-1}}\,\vec{\mathbf{b}}_{|\vec{w}|-1}) \qquad \overline{\mathsf{union}_t\,(j,w,\vec{\mathbf{b}})} := \overline{w}\,\vec{\mathbf{b}} \qquad \overline{\overline{\mathsf{union}}_t\,\vec{\mathbf{b}}} := \vec{\mathbf{b}}$$

The flattened version of the memory tree representing the object s in the previous example is as follows:

$$10000100\ 01000100\ \lightning\lightning\lightning\lightning\lightning\lightning\lightning\lightning\ \lightning\lightning\lightning\lightning\lightning\lightning\lightning\lightning\ (\mathsf{ptr}\ p)_0\ (\mathsf{ptr}\ p)_1\ldots(\mathsf{ptr}\ p)_{31}$$

Definition 5.4.3. *The judgment* $\Gamma,\Delta \vdash w : \tau$ *describes that the* memory tree w has type τ. *It is inductively defined as:*

$$\frac{\Gamma \vdash_\mathsf{b} \tau_\mathsf{b} \qquad \Gamma,\Delta \vdash \vec{\mathbf{b}} \qquad |\vec{\mathbf{b}}| = \mathsf{bitsizeof}_\Gamma\,\tau_\mathsf{b}}{\Gamma,\Delta \vdash \mathsf{base}_{\tau_\mathsf{b}}\,\vec{\mathbf{b}} : \tau_\mathsf{b}} \qquad \frac{\Gamma,\Delta \vdash \vec{w} : \tau \qquad |\vec{w}| = n \neq 0}{\Gamma,\Delta \vdash \mathsf{array}_\tau\,\vec{w} : \tau[n]}$$

$$\frac{\begin{array}{c}\Gamma\,t = \vec{\tau} \qquad \Gamma,\Delta \vdash \vec{w} : \vec{\tau} \\ \forall i\,.\,(\Gamma,\Delta \vdash \vec{\mathbf{b}}_i \qquad \vec{\mathbf{b}}_i \text{ all } \lightning \qquad |\vec{\mathbf{b}}_i| = (\mathsf{fieldbitsizes}_\Gamma\,\vec{\tau})_i - \mathsf{bitsizeof}_\Gamma\,\tau_i)\end{array}}{\Gamma,\Delta \vdash \mathsf{struct}_t\,\overrightarrow{w\mathbf{b}} : \mathsf{struct}\ t}$$

$$\frac{\begin{array}{c}\Gamma\,t = \vec{\tau} \qquad i < |\vec{\tau}| \qquad \Gamma,\Delta \vdash w : \tau_i \qquad \Gamma,\Delta \vdash \vec{\mathbf{b}} \qquad \vec{\mathbf{b}} \text{ all } \lightning \\ \mathsf{bitsizeof}_\Gamma\,(\mathsf{union}\ t) = \mathsf{bitsizeof}_\Gamma\,\tau_i + |\vec{\mathbf{b}}| \qquad \neg\mathsf{unmapped}\,(\overline{w}\,\vec{\mathbf{b}})\end{array}}{\Gamma,\Delta \vdash \mathsf{union}_t\,(i,w,\vec{\mathbf{b}}) : \mathsf{union}\ t}$$

$$\frac{\Gamma\,t = \vec{\tau} \qquad \Gamma,\Delta \vdash \vec{\mathbf{b}} \qquad |\vec{\mathbf{b}}| = \mathsf{bitsizeof}_\Gamma\,(\mathsf{union}\ t)}{\Gamma,\Delta \vdash \overline{\mathsf{union}}_t\,\vec{\mathbf{b}} : \mathsf{union}\ t}$$

Although padding bits should be kept indeterminate (see Section 2.5.2), padding bits are explicitly stored in memory trees for uniformity's sake. The typing judgment ensures that the value of each padding bit is $\lightning$ and that the padding thus only have a permission. Storing a value in padding is a no-op (see Definition 5.4.13).

The side-condition $\neg\mathsf{unmapped}\ (\overline{w}\,\vec{\mathbf{b}})$ in the typing rule for a union $\mathsf{union}_t\ (i, w, \vec{\mathbf{b}})$ of a specified variant ensures canonicity. Unions whose permissions are unmapped cannot be accessed and should therefore be in an unspecified variant. This condition is essential for the separation algebra structure, see Sections 8.1 and 8.3.

Definition 5.4.4. Memories *are defined as:*

$$m \in \mathsf{mem} := \mathsf{index} \to_{\mathsf{fin}} (\mathsf{mtree} \times \mathsf{bool} + \mathsf{type}).$$

Each object $(w,\ \mu)$ in memory is annotated with a Boolean μ to describe whether it has been allocated using `malloc` (in case $\mu = \mathsf{true}$) or as a block scope local, static, or global variable (in case $\mu = \mathsf{false}$). The types of deallocated objects are kept to ensure that dangling pointers (which may remain to exist in memory, but cannot be used) have a unique type.

Definition 5.4.5. *The judgment* $\Gamma, \Delta \vdash m$ *describes that* the memory m is valid. *It is defined as the conjunction of:*

1. *For each* o *and* τ *with* $m\,o = \tau$ *we have:*
   a) $\Delta \vdash o : \tau$, *b)* $\Delta \nvdash o\ \mathsf{alive}$, *and c)* $\Gamma \vdash \tau$.
2. *For each* o, w *and* μ *with* $m\,o = (w,\ \mu)$ *we have:*
   a) $\Delta \vdash o : \tau$, *b)* $\Delta \vdash o\ \mathsf{alive}$, *c)* $\Gamma, \Delta \vdash w : \tau$, *and d) not* $\overline{w}$ *all* $(\emptyset,\ \lightning)$.

The judgment $\Delta \vdash o\ \mathsf{alive}$ *on object identifiers is defined in Definition 5.2.2.*

Definition 5.4.6. *The* minimal memory typing environment $\overline{m} \in \mathsf{memenv}$ *of a memory* m *is defined as:*

$$\overline{m} := \lambda o\,.\begin{cases} (\tau,\ \mathsf{true}) & \text{if } m\,o = \tau \\ (\mathsf{typeof}\ w,\ \mathsf{false}) & \text{if } m\,o = (w,\ \mu) \end{cases}$$

Notation 5.4.7. *We let* $\Gamma \vdash m$ *denote* $\Gamma, \overline{m} \vdash m$.

Many of the conditions of the judgment $\Gamma, \Delta \vdash m$ ensure that the types of m match up with the types in the memory environment Δ (see Definition 5.2.2). One may of course wonder why do we not define the judgment $\Gamma \vdash m$ directly, and even consider typing of a memory in an arbitrary memory environment. Consider:

```
int x = 10, *p = &x;
```

Using an assertion of separation logic we can describe the memory induced by the above program as $\mathtt{x} \mapsto 10 \ast \mathtt{p} \mapsto \&\mathtt{x}$. The separation conjunction $\ast$ describes that the memory can be subdivided into two parts, a part for `x` and another part for `p`. When considering $\mathtt{p} \mapsto \&\mathtt{x}$ in isolation, which is common in separation logic, we have a pointer that refers outside the part itself. This isolated part is thus not typeable by $\Gamma \vdash m$, but it is typeable in the context of a the memory environment corresponding to the whole memory. See also Lemma 8.1.8 on page 148.

In the remaining part of this section we will define various auxiliary operations that will be used to define the memory operations in Section 5.6. We give a summary of the most important auxiliary operations:

$$\begin{array}{rlll}
\mathsf{new}_\Gamma^\gamma : \mathsf{type} \to \mathsf{mtree} & & \text{for} & \gamma : \mathsf{perm} \\
(\_)[\,\_\,]_\Gamma : \mathsf{mem} \to \mathsf{addr} \to \mathsf{option\ mtree} & & & \\
(\_)[\_/f]_\Gamma : \mathsf{mem} \to \mathsf{addr} \to \mathsf{mem} & & \text{for} & f : \mathsf{mtree} \to \mathsf{mtree}
\end{array}$$

Intuitively these are just basic tree operations, but unions make their actual definitions more complicated. The indeterminate memory tree $\mathsf{new}_\Gamma^\gamma\,\tau$ consists of indeterminate bits with permission γ, the lookup operation $m[a]_\Gamma$ yields the memory tree at address a in m, and the alter operation $m[a/f]_\Gamma$ applies the function f to the memory tree at address a in m.

The main delicacy of all of these operations is that we sometimes have to interpret bits as memory trees, or reinterpret memory trees as memory trees of a different type. Most notably, reinterpretation is needed when type-punning is performed:

```
union int_or_short { int x; short y; } u = { .x = 3 };
short z = u.y;
```

This code will reinterpret the bit representation of a memory tree representing an `int` value 3 as a memory tree of type `short`. Likewise:

```
union int_or_short { int x; short y; } u;
((unsigned char*)&u)[0] = 3;
((unsigned char*)&u)[1] = 0;
short z = u.y;
```

Here, we poke some bytes into the object representation of `u`, and interpret these as a memory tree of type `short`.

We have defined the flatten operation $\overline{w}$ that takes a memory tree w and yields its bit representation already in Definition 5.4.2. We now define the operation which goes in opposite direction, called the *unflatten operation.*

Definition 5.4.8. *The* unflatten operation $(\_)_\Gamma^\tau : \mathsf{list\ pbit} \to \mathsf{mtree}$ *is defined as:*

$$\begin{aligned}
(\vec{\mathbf{b}})_\Gamma^{\tau_\mathsf{b}} &:= \mathsf{base}_{\tau_\mathsf{b}}\,\vec{\mathbf{b}} \\
(\vec{\mathbf{b}})_\Gamma^{\tau[n]} &:= \mathsf{array}_\tau\,((\vec{\mathbf{b}}_{[0,\,s)})_\Gamma^\tau \ldots (\vec{\mathbf{b}}_{[(n-1)s,\,ns)})_\Gamma^\tau) \text{ where } s := \mathsf{bitsizeof}_\Gamma\,\tau \\
(\vec{\mathbf{b}})_\Gamma^{\mathsf{struct}\ t} &:= \mathsf{struct}_t \begin{pmatrix} (\vec{\mathbf{b}}_{[0,\,s_0)})_\Gamma^{\tau_0}\,\vec{\mathbf{b}}^{\lightning}_{[s_0,\,z_1)} \\ \ldots \\ (\vec{\mathbf{b}}_{[z_{n-1},\,z_{n-1}+s_{n-1})})_\Gamma^{\tau_{n-1}}\,\vec{\mathbf{b}}^{\lightning}_{[z_{n-1}+s_{n-1},\,z_n)} \end{pmatrix} \\
&\ \text{where } \Gamma\,t = \vec{\tau},\ n := |\vec{\tau}|,\ s_i := \mathsf{bitsizeof}_\Gamma\,\tau_i \text{ and } z_i := \mathsf{bitoffsetof}_\Gamma\,\vec{\tau}\,i \\
(\vec{\mathbf{b}})_\Gamma^{\mathsf{union}\ t} &:= \overline{\mathsf{union}_t}\,\vec{\mathbf{b}}
\end{aligned}$$

Here, the operation $(\_)^{\lightning} : \mathsf{pbit} \to \mathsf{pbit}$ *is defined as* $(x,\,b)^{\lightning} := (x,\,\lightning)$.

In the above definition, the need for $\overline{\mathsf{union}_t}\,\vec{\mathbf{b}}$ memory trees becomes clear. While unflattening a bit sequence as union type, there is no way of knowing which variant of the union the bits constitute. Notice that the operations $\overline{(\_)}$ and $(\_)^{\tau}_{\Gamma}$ are neither left nor right inverses of each other:

- We do not have $(\overline{w})^{\tau}_{\Gamma} = w$ for each w with $\Gamma, \Delta \vdash w : \tau$. Variants of unions are destroyed by flattening w.
- We do not have $\overline{(\vec{\mathbf{b}})^{\tau}_{\Gamma}} = \vec{\mathbf{b}}$ for each $\vec{\mathbf{b}}$ with $|\vec{\mathbf{b}}| = \mathsf{bitsizeof}_{\Gamma}\,\tau$ either. Padding bits become indeterminate due to $(\_)^{\lightning}$ by unflattening.

In Section 5.8 we prove weaker variants of these cancellation properties that are sufficient for proofs about program transformations.

Definition 5.4.9. *Given a permission* $\gamma \in \mathsf{perm}$*, the operation* $\mathsf{new}^{\gamma}_{\Gamma} : \mathsf{type} \to \mathsf{mtree}$ *that yields the indeterminate memory tree is defined as:*

$$\mathsf{new}^{\gamma}_{\Gamma}\,\tau := ((\gamma, \lightning)^{\mathsf{bitsizeof}_{\Gamma}\,\tau})^{\tau}_{\Gamma}.$$

The memory tree $\mathsf{new}^{\gamma}_{\Gamma}\,\tau$ that consists of indeterminate bits with permission γ is used for objects with indeterminate value. We have defined $\mathsf{new}^{\gamma}_{\Gamma}\,\tau$ in terms of the unflattening operation for simplicity's sake. This definition enjoys desirable structural properties such as $\mathsf{new}^{\gamma}_{\Gamma}\,(\tau[n]) = (\mathsf{new}^{\gamma}_{\Gamma}\,\tau)^{n}$.

We will now define the lookup operation $m[a]_{\Gamma}$ that yields the subtree at address a in the memory m. The lookup function is partial, it will fail in case a is end-of-array or violates effective types. We first define the counterpart of lookup on memory trees and then lift it to memories.

Definition 5.4.10. *The* lookup operation on memory trees $(\_)[\_]_{\Gamma} : \mathsf{mtree} \to \mathsf{ref} \to \mathsf{option}\ \mathsf{mtree}$ *is defined as:*

$$\begin{aligned}
w[\varepsilon]_{\Gamma} &:= w \\
(\mathsf{array}_{\tau}\,\vec{w})[(\xhookrightarrow{\tau[n]} i)\,\vec{r}]_{\Gamma} &:= w_i[\vec{r}]_{\Gamma} \\
(\mathsf{struct}_t\,\overrightarrow{w\vec{\mathbf{b}}})[(\xhookrightarrow{\mathsf{struct}\ t} i)\,\vec{r}]_{\Gamma} &:= w_i[\vec{r}]_{\Gamma} \\
(\mathsf{union}_t\,(j, w, \vec{\mathbf{b}}))[(\xhookrightarrow{\mathsf{union}\ t}_q i)\,\vec{r}]_{\Gamma} &:= \begin{cases} w[\vec{r}]_{\Gamma} & \text{if } i = j \\ ((\overline{w}\,\vec{\mathbf{b}})_{[0,\,s)})^{\tau_i}_{\Gamma}\,[\vec{r}]_{\Gamma} & \text{if } i \neq j,\ q = \bullet,\ \mathsf{exclusive}\ (\overline{w}\,\vec{\mathbf{b}}) \\ \bot & \text{if } i \neq j,\ q = \circ \end{cases} \\
&\ \text{where } \Gamma\,t = \vec{\tau} \text{ and } s = \mathsf{bitsizeof}_{\Gamma}\,\tau_i \\
(\overline{\mathsf{union}_t}\,\vec{\mathbf{b}})[(\xhookrightarrow{\mathsf{union}\ t}_q i)\,\vec{r}]_{\Gamma} &:= (\vec{\mathbf{b}}_{[0,\,\mathsf{bitsizeof}_{\Gamma}\,\tau_i)})^{\tau_i}_{\Gamma}\,[\vec{r}]_{\Gamma} \quad \text{if } \Gamma\,t = \vec{\tau},\ \mathsf{exclusive}\ \vec{\mathbf{b}}
\end{aligned}$$

The lookup operation uses the annotations $q \in \{\circ, \bullet\}$ on $\xhookrightarrow{\mathsf{union}\ s}_q i$ to give a formal semantics to the *strict-aliasing restrictions* [ISO12, 6.5.2.3].

- The annotation $q = \bullet$ allows a union to be accessed via a reference whose variant is unequal to the current one. This is called type-punning.
- The annotation $q = \circ$ allows a union to be accessed only via a reference whose variant is equal to the current one. This means, it rules out type-punning.

Failure of type-punning is captured by partiality of the lookup operation. The behavior of type-punning of $\mathsf{union}_t\,(j, w, \vec{\mathbf{b}})$ via a reference to variant i is described by the conversion $((\overline{w}\,\vec{\mathbf{b}})_{[0,\,\mathsf{bitsizeof}_\Gamma\,\tau_i)})^{\tau_i}_\Gamma$. The memory tree w is converted into bits and reinterpreted as a memory tree of type τ_i.

The exclusive conditions in the union cases will be explained in Section 8.1.

Definition 5.4.11. *The* lookup operation on memories $(\_)[\_]_\Gamma : \mathsf{mem} \to \mathsf{addr} \to \mathsf{option\ mtree}$ *is defined as:*

$$m[a]_\Gamma := \begin{cases} ((\overline{w[\mathsf{ref}_\Gamma\,a]_\Gamma})_{[i,\,j)})^{\mathsf{unsigned\ char}}_\Gamma & \text{if } a \text{ is a byte address} \\ w[\mathsf{ref}_\Gamma\,a]_\Gamma & \text{if } a \text{ is not a byte address} \end{cases}$$

provided that $m\,(\mathsf{index}\ a) = (w,\ \mu)$. *In omitted cases the result is* $\bot$. *In this definition we let* $i := \mathsf{byte}_\Gamma\,a \cdot \mathsf{char\_bits}$ *and* $j := (\mathsf{byte}_\Gamma\,a + 1) \cdot \mathsf{char\_bits}$.

We have to take special care of addresses that refer to individual bytes rather than whole objects. Consider:

```
struct S { int x; int y; } s = { .x = 1, .y = 2 };
unsigned char z = ((unsigned char*)&s)[0];
```

In this code, we obtain the first byte `((unsigned char*)&s)[0]` of the struct `s`. This is formalized by flattening the entire memory tree of the struct `s`, and selecting the appropriate byte.

The C11 standard's description of effective types [ISO12, 6.5p6-7] states that an access (which is either a read or store) affects the effective type of the accessed object. This means that although reading from memory does not affect the memory contents, it may still affect the effective types. Let us consider an example where it is indeed the case that effective types are affected by a read:

```
short g(int *p, short *q) {
  short z = *q; *p = 10; return z;
}
int main() {
  union int_or_short { int x; short y; } u;
  // initialize u with zeros, the variant of u remains unspecified
  for (size_t i = 0; i < sizeof(u); i++) ((unsigned char*)&u)[i] = 0;
  return g(&u.x, &u.y);
}
```

In this code, the variant of the union `u` is initially unspecified. The read `*q` in `g` *forces* its variant to `y`, making the assignment `*p` to variant `x` undefined. Note that it is important that we also assign undefined behavior to this example, a compiler may assume `p` and `q` to not alias regardless of how `g` is called.

We factor these side-effects out using a function $\mathsf{force}_\Gamma : \mathsf{addr} \to \mathsf{mem} \to \mathsf{mem}$ that updates the effective types (that is the variants of unions) after a successful lookup. The force_Γ function, as defined in Definition 5.6, can be described in terms of the alter operation $m[a/f]_\Gamma$ that applies the function $f : \mathsf{mtree} \to \mathsf{mtree}$ to the object at address a in the memory m and update variants of unions accordingly to a. To define force_Γ we let f be the identify.

Definition 5.4.12. *Given a function* $f : \mathsf{mtree} \to \mathsf{mtree}$, *the* alter operation on memory trees $(\_)[\_/f]_\Gamma : \mathsf{mtree} \to \mathsf{ref} \to \mathsf{mtree}$ *is defined as:*

$$w[\varepsilon/f]_\Gamma := f\,w$$

$$\begin{aligned}
(\mathsf{array}_\tau\,\vec{w})[(\xhookrightarrow{\tau[n]} i)\,\vec{r}/f]_\Gamma &:= \mathsf{array}_\tau\,(\vec{w}[i := w_i[\vec{r}/f]_\Gamma])\\
(\mathsf{struct}_t\,\overrightarrow{w\vec{\mathbf{b}}})[(\xhookrightarrow{\mathsf{struct}\ t} i)\,\vec{r}/f]_\Gamma &:= \mathsf{struct}_t\,((\overrightarrow{w\vec{\mathbf{b}}})[i := w_i[\vec{r}/f]_\Gamma \vec{\mathbf{b}}_i])\\
(\mathsf{union}_t\,(i, w, \vec{\mathbf{b}}))[(\xhookrightarrow{\mathsf{union}\ t}_q j)\,\vec{r}/f]_\Gamma &:= \begin{cases} \mathsf{union}_t\,(i, w[\vec{r}/f]_\Gamma, \vec{\mathbf{b}}) & \text{if } i = j\\ \mathsf{union}_t\,(i, (((\overline{w}\vec{\mathbf{b}})_{[0,\,s)})_\Gamma^{\tau_i})[\vec{r}/f]_\Gamma, (\overline{w}\vec{\mathbf{b}})^{\lightning}_{[s,\,z)}) & \text{if } i \neq j \end{cases}\\
(\overline{\mathsf{union}_t}\,\vec{\mathbf{b}})[(\xhookrightarrow{\mathsf{union}\ t}_q i)\,\vec{r}/f]_\Gamma &:= \mathsf{union}_t\,(i, ((\vec{\mathbf{b}}_{[0,\,s)})_\Gamma^{\tau_i})[\vec{r}/f]_\Gamma, \vec{\mathbf{b}}^{\lightning}_{[s,\,z)})
\end{aligned}$$

In the last two cases we have $\Gamma\,t = \vec{\tau}$, $s := \mathsf{bitsizeof}_\Gamma\,\tau_i$ *and* $z := \mathsf{bitsizeof}_\Gamma\,(\mathsf{union}\ t)$. *The result of* $w[\vec{r}/f]_\Gamma$ *is only well-defined in case* $w[\vec{r}]_\Gamma \neq \bot$.

Definition 5.4.13. *Given a function* $f : \mathsf{mtree} \to \mathsf{mtree}$, *the* alter operation on memories $(\_)[\_/f]_\Gamma : \mathsf{mem} \to \mathsf{addr} \to \mathsf{mem}$ *is defined as:*

$$m[a/f]_\Gamma := \begin{cases} m[(\mathsf{index}\ a) := (w[\mathsf{ref}_\Gamma\,a/\overline{f}]_\Gamma,\ \mu)] & \text{if } a \text{ is a byte address}\\ m[(\mathsf{index}\ a) := (w[\mathsf{ref}_\Gamma\,a/f]_\Gamma,\ \mu)] & \text{if } a \text{ is not a byte address} \end{cases}$$

provided that $m\,(\mathsf{index}\ a) = (w,\ \mu)$. *In this definition we let:*

$$\overline{f}\,w := (\overline{w}_{[0,\,i)}\ \overline{f\ (\overline{w}_{[i,\,j)})_\Gamma^{\mathsf{unsigned\ char}}}\ \overline{w}_{[j,\,\mathsf{bitsizeof}_\Gamma\,(\mathsf{typeof}\ w))})_\Gamma^{\mathsf{typeof}\ w}$$

where $i := \mathsf{byte}_\Gamma\,a \cdot \mathsf{char\_bits}$ *and* $j := (\mathsf{byte}_\Gamma\,a + 1) \cdot \mathsf{char\_bits}$.

The lookup and alter operation enjoy various properties; they preserve typing and satisfy laws about their interaction. We list some for illustration.

Lemma 5.4.14 (Alter commutes). *If* $\Gamma, \Delta \vdash m$, $a_1 \perp_\Gamma a_2$ *with:*

- $\Gamma, \Delta \vdash a_1 : \tau_1$, $m[a_1]_\Gamma = w_1$, *and* $\Gamma, \Delta \vdash f_1\,w_1 : \tau_1$, *and*
- $\Gamma, \Delta \vdash a_2 : \tau_2$, $m[a_2]_\Gamma = w_2$, *and* $\Gamma, \Delta \vdash f_2\,w_2 : \tau_2$,

then we have:

$$m[a_2/f_2]_\Gamma[a_1/f_1]_\Gamma = m[a_1/f_1]_\Gamma[a_2/f_2]_\Gamma.$$

Lemma 5.4.15. *If* $\Gamma, \Delta \vdash m$, $m[a]_\Gamma = w$, *and* a *is not a byte address, then:*

$$(m[a/f]_\Gamma)[a]_\Gamma = f\,w.$$

A variant of Lemma 5.4.15 for byte addresses is more subtle because a byte address can be used to modify padding. Since modifications of padding are masked, a successive lookup may yield a memory tree with more indeterminate bits. In Section 5.8 we present an alternative lemma that covers this situation.

We conclude this section with a useful helper function that *zips* a memory tree and a list. It is used in for example Definitions 5.6.5 and 8.1.4.

Definition 5.4.16. *Given a function* $f : \mathsf{pbit} \to B \to \mathsf{pbit}$*, the operation that zips the leaves* $\hat{f} : \mathsf{mtree} \to \mathsf{list}\ B \to \mathsf{mtree}$ *is defined as:*

$$\begin{aligned}
\hat{f}\,(\mathsf{base}_{\tau_\mathsf{b}}\ \vec{\mathbf{b}})\ \vec{y} &:= \mathsf{base}_{\tau_\mathsf{b}}\ (f\ \vec{\mathbf{b}}\ \vec{y})\\
\hat{f}\,(\mathsf{array}_\tau\ \vec{w})\ \vec{y} &:= \mathsf{array}_\tau\ (\hat{f}\ w_0\ \vec{y}_{[0,\, s_1)} \ldots \hat{f}\ w_{n-1}\ \vec{y}_{[s_{n-1},\, s_n)})\\
&\ \text{where } n := |\vec{w}| \text{ and } s_i := \Sigma_{j<i}|\overline{w_j}|\\
\hat{f}\,(\mathsf{struct}_t\ \overrightarrow{w\vec{\mathbf{b}}})\ \vec{y} &:= \mathsf{struct}_t\ \begin{pmatrix} \hat{f}\ w_0\ \vec{y}_{[0,\, s_0)}\ \ f\ \vec{\mathbf{b}}_0\ \vec{y}_{[s_0,\, z_1)} \\ \ldots \\ \hat{f}\ w_{n-1}\ \vec{y}_{[z_{n-1},\, z_{n-1}+s_{n-1})}\ \ f\ \vec{\mathbf{b}}_{n-1}\ \vec{y}_{[z_{n-1}+s_{n-1},\, z_n)} \end{pmatrix}\\
&\ \text{where } n := |\vec{w}|,\ s_i := |\overline{w_i}|, \text{ and } z_i := \Sigma_{j<i}|\overline{w_i}\,\vec{\mathbf{b}}_i|\\
\hat{f}\,(\mathsf{union}_t\ (i, w, \vec{\mathbf{b}}))\ \vec{y} &:= \mathsf{union}_t\ (i, \hat{f}\ w\ \vec{y}_{[0,\, |\overline{w}|)}, f\ \vec{\mathbf{b}}\ \vec{y}_{[|\overline{w}|,\, |\overline{w}\,\vec{\mathbf{b}}|)})\\
\hat{f}\,(\overline{\mathsf{union}}_t\ \vec{\mathbf{b}})\ \vec{y} &:= \overline{\mathsf{union}}_t\ (f\ \vec{\mathbf{b}}\ \vec{y})
\end{aligned}$$

5.5 Representation of values

Memory trees (Definition 5.4.1) are still rather low-level and expose permissions and implementation specific properties such as bit representations. In this section we define *abstract values*, which are like memory trees but have mathematical integers and pointers instead of bit representations as leaves. Abstract values are used in the external interface of the memory model (Section 5.1).

Definition 5.5.1. Base values *are inductively defined as:*

$$v_\mathsf{b} \in \mathsf{baseval} ::= \mathsf{indet}\,\tau_\mathsf{b} \mid \mathsf{nothing} \mid \mathsf{int}_{\tau_\mathsf{i}}\,x \mid \mathsf{ptr}\,p \mid \mathsf{byte}\,\vec{b}.$$

While performing byte-wise operations (for example, byte-wise copying a struct containing pointer values), abstraction is broken, and pointer fragment bits have to reside outside of memory. The value $\mathsf{byte}\,\vec{b}$ is used for this purpose.

Definition 5.5.2. *The judgment* $\Gamma, \Delta \vdash_\mathsf{b} v_\mathsf{b} : \tau_\mathsf{b}$ *describes that* the base value v_b has base type τ_b*. It is inductively defined as:*

$$\frac{\Gamma \vdash_\mathsf{b} \tau_\mathsf{b} \qquad \tau_\mathsf{b} \neq \mathsf{void}}{\Gamma, \Delta \vdash_\mathsf{b} \mathsf{indet}\,\tau_\mathsf{b} : \tau_\mathsf{b}} \qquad \frac{}{\Gamma, \Delta \vdash_\mathsf{b} \mathsf{nothing} : \mathsf{void}} \qquad \frac{x : \tau_\mathsf{i}}{\Gamma, \Delta \vdash_\mathsf{b} \mathsf{int}_{\tau_\mathsf{i}}\,x : \tau_\mathsf{i}}$$

$$\frac{\Gamma, \Delta \vdash_* p : \sigma_\mathsf{p}}{\Gamma, \Delta \vdash_\mathsf{b} \mathsf{ptr}\,p : \sigma_\mathsf{p}*} \qquad \frac{\Gamma, \Delta \vdash \vec{b} \quad |\vec{b}| = \mathsf{char\_bits} \quad \textit{Not } \vec{b} \textit{ all in } \{0,1\} \quad \textit{Not } \vec{b} \textit{ all } \lightning}{\Gamma, \Delta \vdash_\mathsf{b} \mathsf{byte}\,\vec{b} : \mathsf{unsigned\ char}}$$

The side-conditions of the typing rule for $\mathsf{byte}\,\vec{b}$ ensure canonicity of representations of base values. It ensures that the construct $\mathsf{byte}\,\vec{b}$ is only used if $\vec{b}$ cannot be represented as an integer $\mathsf{int}_{\mathsf{unsigned\ char}}\,x$ or $\mathsf{indet}\,(\mathsf{unsigned\ char})$.

In Definition 5.5.6 we define abstract values by extending base values with constructs for arrays, structs and unions. In order to define the operations to look up and store values in memory, we define conversion operations between abstract values and memory trees. Recall that the leaves of memory trees, which represent base values, are just sequences of bits. We therefore first define operations that convert base values to and from bits. These operations are called flatten and unflatten.

Definition 5.5.3. *The* flatten operation $\overline{(\_)}^{\Gamma} : \mathsf{baseval} \to \mathsf{list\ bit}$ *is defined as:*

$$\begin{aligned}
\overline{\mathsf{indet}\,\tau_{\mathsf{b}}}^{\Gamma} &:= \lightning^{\mathsf{bitsizeof}_{\Gamma}\,\tau_{\mathsf{b}}}\\
\overline{\mathsf{nothing}}^{\Gamma} &:= \lightning^{\mathsf{bitsizeof}_{\Gamma}\,\mathsf{void}}\\
\overline{\mathsf{int}_{\tau_{\mathsf{i}}}\,x}^{\Gamma} &:= \overline{x : \tau_{\mathsf{i}}}\\
\overline{\mathsf{ptr}\,p}^{\Gamma} &:= (\mathsf{ptr}\,|\,p\,|_{\circ})_0 \dots (\mathsf{ptr}\,|\,p\,|_{\circ})_{\mathsf{bitsizeof}_{\Gamma}\,(\mathsf{typeof}\ p*)-1}\\
\overline{\mathsf{byte}\,\vec{b}}^{\Gamma} &:= \vec{b}
\end{aligned}$$

The operation $\overline{\_ : \tau_{\mathsf{i}}} : \mathbb{Z} \to \mathsf{list\ bool}$ *is defined in Definition 3.1.4 on page 41.*

Definition 5.5.4. *The* unflatten operation $(\_)^{\Gamma}_{\tau_{\mathsf{b}}} : \mathsf{list\ bit} \to \mathsf{baseval}$ *is defined as:*

$$(\vec{b})^{\mathsf{void}}_{\Gamma} := \mathsf{nothing}$$

$$(\vec{b})^{\tau_{\mathsf{i}}}_{\Gamma} := \begin{cases} \mathsf{int}_{\tau_{\mathsf{i}}}\,(\vec{\beta})_{\tau_{\mathsf{i}}} & \text{if } \vec{b} \text{ is a } \{0,1\} \text{ sequence } \vec{\beta}\\ \mathsf{byte}\,\vec{b} & \text{if } \tau_{\mathsf{i}} = \mathsf{unsigned\ char}, \text{ not } \vec{b} \text{ all in } \{0,1\}, \text{ and not } \vec{b} \text{ all } \lightning\\ \mathsf{indet}\,\tau_{\mathsf{i}} & \text{otherwise} \end{cases}$$

$$(\vec{b})^{\sigma_{\mathsf{p}}*}_{\Gamma} := \begin{cases} \mathsf{ptr}\,p & \text{if } \vec{b} = (\mathsf{ptr}\ p)_0 \dots (\mathsf{ptr}\ p)_{\mathsf{bitsizeof}_{\Gamma}\,(\sigma_{\mathsf{p}}*)-1} \text{ and } \mathsf{typeof}\ p = \sigma_{\mathsf{p}}\\ \mathsf{indet}\,(\sigma_{\mathsf{p}}*) & \text{otherwise} \end{cases}$$

The operation $(\_)_{\tau_{\mathsf{i}}} : \mathsf{list\ bool} \to \mathbb{Z}$ *is defined in Definition 3.1.4 on 41.*

The encoding of pointers is an important aspect of the flatten operation related to our treatment of effective types. Pointers are encoded as sequences of *frozen* pointer fragment bits $(\mathsf{ptr}\ |\,p\,|_{\circ})_i$ (see Definition 5.2.4 for the definition of frozen pointers). Recall that the flatten operation is used to store base values in memory, whereas the unflatten operation is used to retrieve them. This means that whenever a pointer p is stored and read back, the frozen variant $|\,p\,|_{\circ}$ is obtained.

Lemma 5.5.5. *For each* $\Gamma, \Delta \vdash_{\mathsf{b}} v_{\mathsf{b}} : \tau_{\mathsf{b}}$ *we have* $(\overline{v_{\mathsf{b}}}^{\Gamma})^{\tau_{\mathsf{b}}}_{\Gamma} = |\,v_{\mathsf{b}}\,|_{\circ}$.

Freezing formally describes the situations in which type-punning is allowed since a frozen pointer cannot be used to access a union of another variant than its current one (Definition 5.4.10). Let us consider an example:

```
union U { int x; short y; } u = { .x = 3 };
short *p = &u.y;  // a frozen version of the pointer &u.y is stored
printf("%d", *p); // type-punning via a frozen pointer -> undefined
```

Here, an attempt to type-punning is performed via the frozen pointer p, which is formally represented as:

$$(o_u : \mathsf{union}\ \mathtt{U}, \xhookrightarrow{\mathsf{union}\ \mathtt{U}}_{\circ} 1, 0)_{\mathsf{signed\ short} >_{*} \mathsf{signed\ short}}.$$

The lookup operation on memory trees (which will be used to obtain the value of `*p` from memory, see Definitions 5.4.10 and 5.6.5) will fail. The annotation $\circ$ prevents a union from being accessed through an address to another variant than its current one. In the example below type-punning is allowed:

```
union U { int x; short y; } u = { .x = 3 };
printf("%d", u.y);
```

Here, type-punning is allowed because it is performed directly via `u.y`, which has not been stored in memory, and thus has not been frozen.

Definition 5.5.6. Abstract values *are inductively defined as:*

$$v \in \mathsf{val} ::= v_{\mathsf{b}} \mid \mathsf{array}_\tau\,\vec{v} \mid \mathsf{struct}_t\,\vec{v} \mid \mathsf{union}_t\,(i, v) \mid \overline{\mathsf{union}}_t\,\vec{v}.$$

The abstract value $\overline{\mathsf{union}}_t\,\vec{v}$ represents a union whose variant is unspecified. The values $\vec{v}$ correspond to interpretations of *all* variants of $\mathsf{union}\ t$. Consider:

```
union U { int x; short y; int *p; } u;
for (size_t i = 0; i < sizeof(u); i++) ((unsigned char*)&u)[i] = 0;
```

Here, the object representation of `u` is initialized with zeros, and its variant thus remains unspecified. The abstract value of `u` is[1]:

$$\overline{\mathsf{union}}_{\mathsf{U}}\,[\,\mathsf{int}_{\mathsf{signed\ int}}\,0,\ \mathsf{int}_{\mathsf{signed\ short}}\,0,\ \mathsf{indet}\,(\mathsf{signed\ int}{*})\,]$$

Recall that the variants of a union occupy a single memory area, so the sequence $\vec{v}$ of a union value $\overline{\mathsf{union}}_t\,\vec{v}$ cannot be arbitrary. There should be a common bit sequence representing it. This is not the case in:

$$\overline{\mathsf{union}}_{\mathsf{U}}\,[\,\mathsf{int}_{\mathsf{signed\ int}}\,0,\ \mathsf{int}_{\mathsf{signed\ short}}\,1,\ \mathsf{indet}\,(\mathsf{signed\ int}{*})\,]$$

The typing judgment for abstract values guarantees that $\vec{v}$ can be represented by a common bit sequence. In order to express this property, we first define the unflatten operation that converts a bit sequence into an abstract value.

Definition 5.5.7. *The* unflatten operation $(\_)^{\tau}_{\Gamma} : \mathsf{list\ bit} \to \mathsf{val}$ *is defined as:*

$$\begin{aligned}
(\vec{b})^{\tau_{\mathsf{b}}}_{\Gamma} &:= (\vec{b})^{\tau_{\mathsf{b}}}_{\Gamma} && \text{(the right hand side is Definition 5.5.3 on base values)}\\
(\vec{b})^{\tau[n]}_{\Gamma} &:= \mathsf{array}_\tau\,((\vec{b}_{[0,\,s)})^{\tau}_{\Gamma} \dots (\vec{b}_{[(n-1)s,\,ns)})^{\tau}_{\Gamma}) \text{ where } s := \mathsf{bitsizeof}_\Gamma\,\tau\\
(\vec{b})^{\mathsf{struct}\ t}_{\Gamma} &:= \mathsf{struct}_t\,((\vec{b}_{[0,\,s_0)})^{\tau_0}_{\Gamma} \dots (\vec{b}_{[z_{n-1},\,z_{n-1}+s_{n-1})})^{\tau_{n-1}}_{\Gamma})\\
&\quad\ \text{where } \Gamma\,t = \vec{\tau},\ n := |\vec{\tau}|,\ s_i := \mathsf{bitsizeof}_\Gamma\,\tau_i \text{ and } z_i := \mathsf{bitoffsetof}_\Gamma\,\vec{\tau}\ i\\
(\vec{b})^{\mathsf{union}\ t}_{\Gamma} &:= \overline{\mathsf{union}}_t\,((\vec{b}_{[0,\,s_0)})^{\tau_0}_{\Gamma} \dots (\vec{b}_{[0,\,s_{n-1})})^{\tau_{n-1}}_{\Gamma})\\
&\quad\ \text{where } \Gamma\,t = \vec{\tau},\ n := |\vec{\tau}| \text{ and } s_i := \mathsf{bitsizeof}_\Gamma\,\tau_i
\end{aligned}$$

[1] Note that the C11 standard does not guarantee that the `NULL` pointer is represented as zeros, thus `u.p` is not necessarily `NULL`.

Definition 5.5.8. *The judgment* $\Gamma, \Delta \vdash v : \tau$ *describes that* the value v has type τ. *It is inductively defined as:*

$$\frac{\Gamma, \Delta \vdash_{\mathsf{b}} v_{\mathsf{b}} : \tau_{\mathsf{b}}}{\Gamma, \Delta \vdash v_{\mathsf{b}} : \tau_{\mathsf{b}}} \qquad \frac{\Gamma, \Delta \vdash \vec{v} : \tau \qquad |\vec{v}| = n \neq 0}{\Gamma, \Delta \vdash \mathsf{array}_{\tau}\, \vec{v} : \tau[n]}$$

$$\frac{\Gamma\, t = \vec{\tau} \qquad \Gamma, \Delta \vdash \vec{v} : \vec{\tau}}{\Gamma, \Delta \vdash \mathsf{struct}_t\, \vec{v} : \mathsf{struct}\ t} \qquad \frac{\Gamma\, t = \vec{\tau} \qquad i < |\vec{\tau}| \qquad \Gamma, \Delta \vdash v : \tau_i}{\Gamma, \Delta \vdash \mathsf{union}_t\, (i, v) : \mathsf{union}\ t}$$

$$\frac{\Gamma\, t = \vec{\tau} \qquad \Gamma, \Delta \vdash \vec{v} : \vec{\tau} \qquad \Gamma, \Delta \vdash \vec{b} \qquad \forall i\, .\, (v_i = (\vec{b}_{[0,\, \mathsf{bitsizeof}_{\Gamma}\, \tau_i)})^{\tau_i}_{\Gamma})}{\Gamma, \Delta \vdash \overline{\mathsf{union}}_t\, \vec{v} : \mathsf{union}\ t}$$

The flatten operation $\overline{(\_)}^{\Gamma} : \mathsf{val} \to \mathsf{list\ bit}$, which converts an abstract value v into a bit representation $\overline{v}^{\Gamma}$, is more difficult to define (we need this operation to define the conversion operation from abstract values into memory trees, see Definition 5.5.11). Since padding bits are not present in abstract values, we have to insert these. Also, in order to obtain the bit representation of an unspecified $\overline{\mathsf{union}}_t\, \vec{v}$ value, we have to *construct* the common bit sequence $\vec{b}$ representing $\vec{v}$. The typing judgment guarantees that such a sequence exists, but since it is not explicit in the value $\overline{\mathsf{union}}_t\, \vec{v}$, we have to reconstruct it from $\vec{v}$. Consider:

```
union U { struct S { short y; void *p; } x1; int x2; };
```

Assuming $\mathsf{sizeof}_{\Gamma}\ (\mathsf{signed\ int}) = \mathsf{sizeof}_{\Gamma}\ (\mathsf{any}{*}) = 4$ and $\mathsf{sizeof}_{\Gamma}\ (\mathsf{signed\ short}) = 2$, a well-typed union U value of an unspecified variant may be:

$$v = \overline{\mathsf{union}}_{\mathtt{U}}\, [\, \mathsf{struct}_{\mathtt{S}}\, [\, \mathsf{int}_{\mathsf{signed\ short}}\, 0, \mathsf{ptr}\, p\,], \mathsf{int}_{\mathsf{signed\ int}}\, 0\,].$$

The flattened versions of the variants of v are:

$$\begin{array}{r@{\;}l}
\overline{\mathsf{struct}_{\mathtt{S}}\, [\, \mathsf{int}_{\mathsf{signed\ short}}\, 0, \mathsf{ptr}\, p\,]}^{\Gamma} & = 0 \ldots 0\, 0 \ldots 0\, \lightning \ldots \lightning\, \lightning \ldots \lightning\ (\mathsf{ptr}\ p)_0 \ldots (\mathsf{ptr}\ p)_{31} \\
\overline{\mathsf{int}_{\mathsf{signed\ int}}\, 0}^{\Gamma} & = 0 \ldots 0\, 0 \ldots 0\, 0 \ldots 0\, 0 \ldots 0 \\
\hline
\overline{v}^{\Gamma} & = 0 \ldots 0\, 0 \ldots 0\, 0 \ldots 0\, 0 \ldots 0\ (\mathsf{ptr}\ p)_0 \ldots (\mathsf{ptr}\ p)_{31}
\end{array}$$

This example already illustrates that so as to obtain the common bit sequence $\overline{v}^{\Gamma}$ of v we have to insert padding bits and "join" the padded bit representations.

Definition 5.5.9. *The* join operation on bits $\sqcup : \mathsf{bit} \to \mathsf{bit} \to \mathsf{bit}$ *is defined as:*

$$\lightning \sqcup b := b \qquad b \sqcup \lightning := b \qquad b \sqcup b := b.$$

Definition 5.5.10. *The* flatten operation $\overline{(\_)}^{\Gamma} : \mathsf{val} \to \mathsf{list\ bit}$ *is defined as:*

$$\begin{aligned}
\overline{v_{\mathsf{b}}}^{\Gamma} &:= \overline{v_{\mathsf{b}}}^{\Gamma} \\
\overline{\mathsf{array}_{\tau}\, \vec{v}}^{\Gamma} &:= \overline{v_0}^{\Gamma} \ldots \overline{v_{|\vec{v}|-1}}^{\Gamma} \\
\overline{\mathsf{struct}_t\, \vec{v}}^{\Gamma} &:= (\overline{v_0}^{\Gamma} \lightning^{\infty})_{[0,\, z_0)} \ldots (\overline{v_{n-1}}^{\Gamma} \lightning^{\infty})_{[0,\, z_{n-1})} \\
&\qquad \text{where } \Gamma\, t = \vec{\tau},\ n := |\vec{\tau}|, \text{ and } z_i := \mathsf{bitoffsetof}_{\Gamma}\, \vec{\tau}\, i \\
\overline{\mathsf{union}_t\, (i, v)}^{\Gamma} &:= (\overline{v}^{\Gamma} \lightning^{\infty})_{[0,\, \mathsf{bitsizeof}_{\Gamma}\ (\mathsf{union}\ t))} \\
\overline{\overline{\mathsf{union}}_t\, \vec{v}}^{\Gamma} &:= \textstyle\bigsqcup_{i=0}^{|\vec{v}|-1} (\overline{v_i}^{\Gamma} \lightning^{\infty})_{[0,\, \mathsf{bitsizeof}_{\Gamma}\ (\mathsf{union}\ t))}
\end{aligned}$$

The operation $\mathsf{ofval}_\Gamma : \mathsf{list\ perm} \to \mathsf{val} \to \mathsf{mtree}$, which converts a value v of type τ into a memory tree $\mathsf{ofval}_\Gamma\,\vec{\gamma}\,v$, is albeit technical fairly straightforward. In principle it is just a recursive definition that uses the flatten operation $\overline{v_\mathsf{b}}^{\Gamma}$ for base values v_b and the flatten operation $\overline{\overline{\mathsf{union}_t\,\vec{v}}}^{\Gamma}$ for unions $\overline{\mathsf{union}_t}\,\vec{v}$ of an unspecified variant.

The technicality is that abstract values do not contain permissions, so we have to merge the given value with permissions. The sequence $\vec{\gamma}$ with $|\vec{\gamma}| = \mathsf{bitsizeof}_\Gamma\,\tau$ represents a flattened sequence of permissions. In the definition of the memory store $m\langle a := v\rangle_\Gamma$ (see Definition 5.6.5), we convert v into the stored memory tree $\mathsf{ofval}_\Gamma\,\vec{\gamma}\,v$ where γ constitutes the old permissions of the object at address a.

Definition 5.5.11. *The operation* $\mathsf{ofval}_\Gamma : \mathsf{list\ perm} \to \mathsf{val} \to \mathsf{mtree}$ *is defined as:*

$$\begin{aligned}
\mathsf{ofval}_\Gamma\,\vec{\gamma}\,(v_\mathsf{b}) &:= \mathsf{base}_{\mathsf{typeof}\ v_\mathsf{b}}\,\overrightarrow{\gamma b} \quad \text{where } \vec{b} := \overline{v_\mathsf{b}}^{\Gamma}\\
\mathsf{ofval}_\Gamma\,\vec{\gamma}\,(\mathsf{array}_\tau\,\vec{v}) &:= \mathsf{array}_\tau\,(\mathsf{ofval}_\Gamma\,\vec{\gamma}_{[0,\,s)}\,v_0 \ldots \mathsf{ofval}_\Gamma\,\vec{\gamma}_{[(n-1)s,\,ns)}\,v_{n-1})\\
&\ \text{where } s := \mathsf{bitsizeof}_\Gamma\,\tau \text{ and } n := |\vec{v}|\\
\mathsf{ofval}_\Gamma\,\vec{\gamma}\,(\mathsf{struct}_t\,\vec{v}) &:= \mathsf{struct}_t \begin{pmatrix} \mathsf{ofval}_\Gamma\,\vec{\gamma}_{[0,\,s_0)}\,v_0\ \ \vec{\gamma}^{\lightning}_{[s_0,\,z_1)}\\ \ldots\\ \mathsf{ofval}_\Gamma\,\vec{\gamma}_{[z_{n-1},\,z_{n-1}+s_{n-1})}\,v_{n-1}\ \ \vec{\gamma}^{\lightning}_{[z_{n-1}+s_{n-1},\,z_n)} \end{pmatrix}\\
&\ \text{where } \Gamma\,t = \vec{\tau},\ n := |\vec{\tau}|,\ s_i := \mathsf{bitsizeof}_\Gamma\,\tau_i\\
&\ \text{and } z_i := \mathsf{bitoffsetof}_\Gamma\,\vec{\tau}\,i\\
\mathsf{ofval}_\Gamma\,\vec{\gamma}\,(\mathsf{union}_t\,(i, v)) &:= \mathsf{union}_t\,(i, \mathsf{ofval}_\Gamma\,\vec{\gamma}_{[0,\,s)}\,v, \vec{\gamma}^{\lightning}_{[s,\,\mathsf{bitsizeof}_\Gamma\,(\mathsf{union}\ t))})\\
&\ \text{where } s := \mathsf{bitsizeof}_\Gamma\,(\mathsf{typeof}\ v)\\
\mathsf{ofval}_\Gamma\,\vec{\gamma}\,(\overline{\mathsf{union}_t}\,\vec{v}) &:= \overline{\mathsf{union}_t}\,\overrightarrow{\gamma b} \quad \text{where } \vec{b} := \overline{\overline{\mathsf{union}_t}\,\vec{v}}^{\Gamma}
\end{aligned}$$

Converting a memory tree into a value is as expected: permissions are removed and unions are interpreted as values corresponding to each variant.

Definition 5.5.12. *The operation* $\mathsf{toval}_\Gamma : \mathsf{mtree} \to \mathsf{val}$ *is defined as:*

$$\begin{aligned}
\mathsf{toval}_\Gamma\,(\mathsf{base}_{\tau_\mathsf{b}}\,\overrightarrow{\gamma b}) &:= (\vec{b})^{\tau_\mathsf{b}}_{\Gamma}\\
\mathsf{toval}_\Gamma\,(\mathsf{array}_\tau\,\vec{w}) &:= \mathsf{array}_\tau\,(\mathsf{toval}_\Gamma\,w_0 \ldots \mathsf{toval}_\Gamma\,w_{|\vec{w}|-1})\\
\mathsf{toval}_\Gamma\,(\mathsf{struct}_t\,\overrightarrow{w\mathbf{b}}) &:= \mathsf{struct}_t\,(\mathsf{toval}_\Gamma\,w_0 \ldots \mathsf{toval}_\Gamma\,w_{|\vec{w}|-1})\\
\mathsf{toval}_\Gamma\,(\mathsf{union}_t\,(i, w, \vec{\mathbf{b}})) &:= \mathsf{union}_t\,(i, \mathsf{toval}_\Gamma\,w)\\
\mathsf{toval}_\Gamma\,(\overline{\mathsf{union}_t}\,\overrightarrow{\gamma b}) &:= (\vec{b})^{\mathsf{union}\ t}_{\Gamma}
\end{aligned}$$

The function toval_Γ is an inverse of ofval_Γ up to freezing of pointers. Freezing is intended, it makes indirect type-punning illegal.

Lemma 5.5.13. *Given* $\Gamma, \Delta \vdash v : \tau$*, and let* $\vec{\gamma}$ *be a flattened sequence of permissions with* $|\vec{\gamma}| = \mathsf{bitsizeof}_\Gamma\,\tau$*, then we have:*

$$\mathsf{toval}_\Gamma\,(\mathsf{ofval}_\Gamma\,\vec{\gamma}\,v) = |\,v\,|_\circ.$$

The other direction does not hold because invalid bit representations will become indeterminate values.

```
struct S { int *p; } s;
for (size_t i = 0; i < sizeof(s); i++) ((unsigned char*)&s)[i] = i;
// s has some bit representation that does not constitute a pointer
struct S s2 = s;
// After reading s, and storing it, there are no guarantees about s2,
// whose object representation thus consists of ↯s
```

We finish this section by defining the indeterminate abstract value $\mathsf{new}_\Gamma\,\tau$, which consists of indeterminate base values. The definition is similar to its counterpart on memory trees (Definition 5.4.9).

Definition 5.5.14. *The operation* $\mathsf{new}_\Gamma : \mathsf{type} \to \mathsf{val}$ *that yields the indeterminate value is defined as:*

$$\mathsf{new}_\Gamma\,\tau := (\lightning^{\mathsf{bitsizeof}_\Gamma\,\tau})^\tau_\Gamma.$$

Lemma 5.5.15. *If* $\Gamma \vdash \tau$*, then:*

$$\mathsf{toval}_\Gamma\,(\mathsf{new}^\gamma_\Gamma\,\tau) = \mathsf{new}_\Gamma\,\tau \quad \text{and} \quad \mathsf{ofval}_\Gamma\,(\gamma^{\mathsf{bitsizeof}_\Gamma\,\tau})\,(\mathsf{new}_\Gamma\,\tau) = \mathsf{new}^\gamma_\Gamma\,\tau.$$

5.6 Memory operations

Now that we have all primitive definitions in place, we can compose these to implement the actual memory operations as described in the beginning of this section. The last part that is missing is a data structure to keep track of objects that have been locked. Intuitively, this data structure should represent a set of addresses, but up to overlapping addresses.

Definition 5.6.1. Locksets *are defined as:*

$$\Omega \in \mathsf{lockset} := \mathcal{P}_{\mathsf{fin}}(\mathsf{index} \times \mathbb{N}).$$

Elements of locksets are pairs $(o,\, i)$ where $o \in \mathsf{index}$ describes the object identifier and $i \in \mathbb{N}$ a bit-offset in the object described by o. We introduce a typing judgment to describe that the structure of locksets matches up with the memory layout.

Definition 5.6.2. *The judgment* $\Gamma, \Delta \vdash \Omega$ *describes that* the lockset Ω is valid*. It is inductively defined as:*

$$\frac{\textit{for each} \quad (o,\, i) \in \Omega \quad \text{there is a } \tau \text{ with} \quad \Delta \vdash o : \tau \text{ and } i < \mathsf{bitsizeof}_\Gamma\,\tau}{\Gamma, \Delta \vdash \Omega}$$

Definition 5.6.3. *The* singleton lockset $\{\_\}_\Gamma : \mathsf{addr} \to \mathsf{lockset}$ *is defined as:*

$$\{a\}_\Gamma := \{(\mathsf{index}\ a, i) \mid \mathsf{bitoffset}_\Gamma\,a \le i < \mathsf{bitoffset}_\Gamma\,a + \mathsf{bitsizeof}_\Gamma\,(\mathsf{typeof}\ a)\}.$$

Lemma 5.6.4. *If* $\Gamma, \Delta \vdash a_1 : \sigma_1$ *and* $\Gamma, \Delta \vdash a_2 : \sigma_2$ *and* $\Gamma \vdash \{a_1, a_2\}$ strict*, then:*

$$a_1 \perp_\Gamma a_2 \quad \text{implies} \quad \{a_1\}_\Gamma \cap \{a_2\}_\Gamma = \emptyset.$$

Definition 5.6.5. *The* memory operations *are defined as:*

$$\begin{aligned}
m\langle a\rangle_\Gamma &:= \mathsf{toval}_\Gamma\, w \quad \text{ if } m[a]_\Gamma = w \text{ and } \forall i\,.\,\mathsf{Readable} \subseteq \mathsf{kind}\ (\overline{w})_i \\
\mathsf{force}_\Gamma\, a\ m &:= m[(\mathsf{index}\ a) := (w[\mathsf{ref}_\Gamma\, a/\lambda w'\,.\,w']_\Gamma,\ \mu)] \quad \text{ if } m\,(\mathsf{index}\ a) = (w,\ \mu) \\
m\langle a := v\rangle_\Gamma &:= m[a/\lambda w\,.\,\mathsf{ofval}_\Gamma\,(\overline{w}_{\mathbf{1}})\,v]_\Gamma \\
\mathsf{writable}_\Gamma\, a\ m &:= \exists w\,.\,m[a]_\Gamma = w \text{ and } \forall i\,.\,\mathsf{Writable} \subseteq \mathsf{kind}\ (\overline{w})_i \\
\mathsf{lock}_\Gamma\, a\ m &:= m[a/\lambda w\,.\ \text{apply } \mathsf{lock} \text{ to all permissions of } w]_\Gamma \\
\mathsf{unlock}\ \Omega\ m &:= \{(o,\ (\hat{f}\ w\ \vec{y},\ \mu)) \mid m\,o = (w,\ \mu)\} \cup \{(o,\ \tau) \mid m\,o = \tau\} \\
&\qquad \text{where } f\ (\gamma,\ b)\,\mathsf{true} := (\mathsf{unlock}\ \gamma,\ b) \\
&\qquad \qquad\ \ f\ (\gamma,\ b)\,\mathsf{false} := (\gamma,\ b), \\
&\qquad \text{and } \vec{y} := ((o,\ 0) \in \Omega) \ldots ((o,\ |\mathsf{bitsizeof}_\Gamma\ (\mathsf{typeof}\ w)| - 1) \in \Omega) \\
\mathsf{alloc}_\Gamma\ o\ v\ \mu\ m &:= m[o := (\mathsf{ofval}_\Gamma\ (\Diamond(0,\ 1)^{\mathsf{bitsizeof}_\Gamma\ (\mathsf{typeof}\ v)})\ v,\ \mu)] \\
\mathsf{freeable}\ a\ m &:= \exists o\,\tau\,\sigma\,n\,w\,.\,a = (o : \tau, \xhookrightarrow{\tau[n]} 0, 0)_{\tau >_* \sigma},\ \ m\,o = (w,\ \mathsf{true}) \\
&\qquad\qquad\qquad \text{and all } \overline{w} \text{ have the permission } \Diamond(0,\ 1) \\
\mathsf{free}\ o\ m &:= m[o := \mathsf{typeof}\ w] \quad \text{ if } m\,o = (w,\ \mu)
\end{aligned}$$

The lookup operation $m\langle a\rangle_\Gamma$ uses the lookup operation $m[a]_\Gamma$ that yields a memory tree w (Definition 5.4.11), and then converts w into the value $\mathsf{toval}_\Gamma\, w$. The operation $m[a]_\Gamma$ already yields $\bot$ in case effective types are violated or a is an end-of-array address. The additional condition of $m\langle a\rangle_\Gamma$ ensures that the permissions allow for a read access. Performing a lookup affects the effective types of the object at address a. This is factored out by the operation $\mathsf{force}_\Gamma\, a\ m$ which applies the identity function to the subobject at address a in the memory m. Importantly, this does not change the memory contents, but merely changes the variants of the involved unions.

The store operation $m\langle a := v\rangle_\Gamma$ uses the alter operation $m[a/\lambda w\,.\,\mathsf{ofval}_\Gamma\,(\overline{w}_{\mathbf{1}})\,v]_\Gamma$ on memories (Definition 5.4.13) to apply $\lambda w\,.\,\mathsf{ofval}_\Gamma\,(\overline{w}_{\mathbf{1}})\,v$ to the subobject at address a. The stored value v is converted into a memory tree while retaining the permissions $\overline{w}_{\mathbf{1}}$ of the previously stored memory tree w at address a.

The definition of $\mathsf{lock}_\Gamma\, a\ m$ is straightforward. In the Coq development we use a map operation on memory trees to apply the function lock (Definition 4.3.2 on page 54) to the permission of each bit of the memory tree at address a.

The operation $\mathsf{unlock}\ \Omega\ m$ unlocks a whole lockset Ω, rather than an individual address, in memory m. For each memory tree w at object identifier o, it converts Ω to a Boolean vector $\vec{y} = ((o,\ 0) \in \Omega) \ldots ((o,\ |\mathsf{bitsizeof}_\Gamma\ (\mathsf{typeof}\ w)| - 1) \in \Omega)$ and merges w with $\vec{y}$ (using Definition 5.4.16) to apply unlock (Definition 4.3.2 on page 54) to the permissions of bits that should be unlocked in w. We show some lemmas to illustrate that the operations for locking and unlocking enjoy the intended behavior:

Lemma 5.6.6. *If* $\Gamma, \Delta \vdash m$ *and* $\Gamma, \Delta \vdash a : \tau$ *and* $\mathsf{writable}_\Gamma\, a\ m$*, then we have:*

$$\mathsf{locks}\ (\mathsf{lock}_\Gamma\ a\ m) = \mathsf{locks}\ m \cup \{a\}_\Gamma.$$

Lemma 5.6.7. *If* $\Omega \subseteq \mathsf{locks}\ m$*, then* $\mathsf{locks}\ (\mathsf{unlock}\ \Omega\ m) = \mathsf{locks}\ m \setminus \Omega$*.*

Provided $o \notin \mathsf{dom}\ m$, allocation $\mathsf{alloc}_\Gamma\ o\ v\ \mu\ m$ extends the memory with a new object holding the value v and *full* permissions $\Diamond(0,1)$. Typically we use $v = \mathsf{new}_\Gamma\ \tau$ for some τ, but global and static variables are allocated with a specific value v.

The operation $\mathsf{free}\ o\ m$ deallocates the object o in m, and keeps track of the type of the deallocated object. In order to deallocate dynamically obtained memory via `free`, the side-condition $\mathsf{freeable}\ a\ m$ describes that the permissions are sufficient for deallocation, and that a points to the first element of a `malloc`ed array.

All operations preserve typing and satisfy the expected laws about their interaction. We list some for illustration.

Fact 5.6.8. *If* $\mathsf{writable}_\Gamma\ a\ m$*, then there exists a value* v *with* $a\langle m\rangle_\Gamma = v$*.*

Lemma 5.6.9 (Stores commute)**.** *If* $\Gamma, \Delta \vdash m$ *and* $a_1 \perp_\Gamma a_2$ *with:*

- $\Gamma, \Delta \vdash a_1 : \tau_1$, $\mathsf{writable}_\Gamma\ a_1\ m$*, and* $\Gamma, \Delta \vdash v_1 : \tau_1$*, and*
- $\Gamma, \Delta \vdash a_2 : \tau_2$, $\mathsf{writable}_\Gamma\ a_2\ m$*, and* $\Gamma, \Delta \vdash v_2 : \tau_2$*,*

then we have:

$$m\langle a_2 := v_2\rangle_\Gamma\langle a_1 := v_1\rangle_\Gamma = m\langle a_1 := v_1\rangle_\Gamma\langle a_2 := v_2\rangle_\Gamma.$$

Lemma 5.6.10 (Looking up after storing)**.** *If* $\Gamma, \Delta \vdash m$ *and* $\Gamma, \Delta \vdash a : \tau$ *and* $\Gamma, \Delta \vdash v : \tau$ *and* $\mathsf{writable}_\Gamma\ a\ m$ *and* a *is not a byte address, then we have:*

$$(m\langle a := v\rangle_\Gamma)\langle a\rangle_\Gamma = |\, v \,|_\circ.$$

Storing a value v in memory and then retrieving it, does not necessarily yield the same value v. It intentionally yields the value $|\, v \,|_\circ$ whose pointers have been frozen. Note that the above result does not hold for byte addresses, which may store a value in a padding byte, in which case the resulting value is indeterminate.

Lemma 5.6.11 (Stores and lookups commute)**.** *If* $\Gamma, \Delta \vdash m$ *and* $a_1 \perp_\Gamma a_2$ *and* $\Gamma, \Delta \vdash a_2 : \tau_2$ *and* $\mathsf{writable}_\Gamma\ a_2\ m$ *and* $\Gamma, \Delta \vdash v_2 : \tau_2$*, then we have:*

$$m\langle a_1\rangle_\Gamma = v_1 \quad \text{implies} \quad (m\langle a_2 := v_2\rangle_\Gamma)\langle a_1\rangle_\Gamma = v_1.$$

These results follow from Lemma 5.4.14, 5.4.15 and 5.5.13.

5.7 Type-based alias analysis

The purpose of C11's notion of effective types [ISO12, 6.5p6-7] is to make it possible for compilers to perform typed-based alias analysis. Consider:

```
short g(int *p, short *q) {
  short x = *q; *p = 10; return x;
}
```

Here, a compiler should be able to assume that `p` and `q` are not aliased because they point to objects with different types (although the integer types $\textsf{signed short}$ and $\textsf{signed int}$ may have the same representation, they have different integer ranks, see Definition 3.1.2 on page 40, and are thus different types). If `g` is called with aliased pointers, execution of the function body should have undefined behavior in order to allow a compiler to soundly assume that `p` and `q` are not aliased.

From the C11 standard's description of effective types it is not immediate that calling `g` with aliased pointers results in undefined behavior. We prove an abstract property of our memory model that shows that this is indeed a consequence, and that indicates a compiler can perform type-based alias analysis. This also shows that our interpretation of effective types of the C11 standard, in line with the interpretation from the GCC documentation [GCC], is sensible.

Definition 5.7.1. *A type τ is a* subobject type of σ*, notation $\tau \subseteq_\Gamma \sigma$, if there exists some reference $\vec{r}$ with $\Gamma \vdash \vec{r} : \sigma \rightarrowtail \tau$.*

For example, `int[2]` is a subobject type of `struct S { int x[2]; int y[3]; }` and `int[2][2]`, but not of `struct S { short x[2]; }`, nor of `int(*)[2]`.

Theorem 5.7.2 (Strict-aliasing)**.** *Given $\Gamma, \Delta \vdash m$, frozen addresses a_1 and a_2 with $\Delta, m \vdash a_1 : \sigma_1$ and $\Delta, m \vdash a_2 : \sigma_2$ and $\sigma_1, \sigma_2 \neq \textsf{unsigned char}$, then either:*

1. *We have $\sigma_1 \subseteq_\Gamma \sigma_2$ or $\sigma_2 \subseteq_\Gamma \sigma_1$.*
2. *We have $a_1 \perp_\Gamma a_2$.*
3. *Accessing a_1 after accessing a_2 and vice versa fails. That means:*
   a) $(\mathsf{force}_\Gamma\, a_2\, m)\langle a_1\rangle_\Gamma = \bot$ and $(\mathsf{force}_\Gamma\, a_1\, m)\langle a_2\rangle_\Gamma = \bot$, and
   b) $m\langle a_2 := v_1\rangle_\Gamma\langle a_1\rangle_\Gamma = \bot$ and $m\langle a_1 := v_2\rangle_\Gamma\langle a_2\rangle_\Gamma = \bot$ for all stored values v_1 and v_2.

This theorem implies that accesses to addresses of disjoint type are either non-overlapping or have undefined behavior. Fact 5.6.8 accounts for a store after a lookup. Using this theorem, a compiler can optimize the generated code in the example based on the assumption that `p` and `q` are not aliased. Reconsider:

```
short g(int *p, short *q) { short x = *q; *p = 10; return x; }
```

If `p` and `q` are aliased, then calling `g` yields undefined behavior because the assignment `*p = 10` violates effective types. Let m be the initial memory while executing `g`, and let $a_{\mathtt{p}}$ and $a_{\mathtt{q}}$ be the addresses corresponding to `p` and `q`, then the condition $\mathsf{writable}_\Gamma\, a_{\mathtt{p}}\, (\mathsf{force}_\Gamma\, a_{\mathtt{q}}\, m)$ does not hold by Theorem 5.7.2 and Fact 5.6.8.

5.8 Memory refinements

This section defines the notion of *memory refinements* that allows us to relate memory states, and in Section 6.7 we prove that the operational semantics is invariant under this notion. Memory refinements form a general way to validate many common-sense properties of the memory model in a formal way. For example, they show that the memory is invariant under relabeling. More interestingly, they show that symbolic information (such as variants of unions) cannot be observed.

Memory refinements also open the door to reason about program transformations. We demonstrate their usage by proving soundness of constant propagation and by verifying an abstract version of `memcpy`.

Memory refinements are a variant of Leroy and Blazy's notion of memory extensions and injections [LB08]. A memory refinement is a relation $m_1 \sqsubseteq_{\Gamma}^{f} m_2$ between a source memory state m_1 and target memory state m_2, where:

1. The function $f : \mathsf{index} \to \mathsf{option}\ (\mathsf{index} \times \mathsf{ref})$ is used to rename object identifiers and to coalesce multiple objects into subobjects of a compound object.
2. Deallocated objects in m_1 may be replaced by arbitrary objects in m_2.
3. Indeterminate bits $\text{\textlightning}$ in m_1 may be replaced by arbitrary bits in m_2.
4. Pointer fragment bits $(\mathsf{ptr}\ p)_i$ that belong to deallocated pointers in m_1 may be replaced by arbitrary bits in m_2.
5. Effective types may be weakened. That means, unions with a specific variant in m_1 may be replaced by unions with an unspecified variant in m_2, and pointers with frozen union annotations $\circ$ in m_1 may be replaced by pointers with unfrozen union annotations $\bullet$ in m_2.

The key property of a memory refinement $m_1 \sqsubseteq_{\Gamma}^{f} m_2$, as well as of Leroy and Blazy's memory extensions and injections, is that memory operations are more defined on the target memory m_2 than on the source memory m_1. For example, if a lookup succeeds on m_1, it also succeed on m_2 and yield a related value.

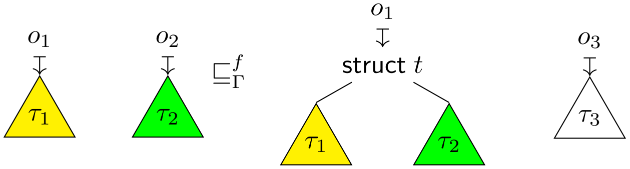
The main judgment $m_1 \sqsubseteq_{\Gamma}^{f:\Delta_1 \mapsto \Delta_2} m_2$ of memory refinements will be built using a series of refinement relations on the structures out of which the memory consists (addresses, pointers, bits, memory trees, values). All of these judgments should satisfy some basic properties, which are captured by the judgment $\Delta_1 \sqsubseteq_{\Delta}^{f} \Delta_2$.

Definition 5.8.1. *A* renaming function $f : \mathsf{index} \to \mathsf{option}\ (\mathsf{index} \times \mathsf{ref})$ *is a* refinement*, notation* $\Delta_1 \sqsubseteq_{\Delta}^{f} \Delta_2$*, if the following conditions hold:*

1. *If* $f\,o_1 = (o, \vec{r_1})$ *and* $f\,o_2 = (o, \vec{r_2})$*, then* $o_1 = o_2$ *or* $\vec{r_1} \perp \vec{r_2}$ *(*injectivity*).*
2. *If* $f\,o_1 = (o_2, \vec{r})$*, then* $\mathsf{frozen}\ \vec{r}$*.*
3. *If* $f\,o_1 = (o_2, \vec{r})$ *and* $\Delta_1 \vdash o_1 : \sigma$*, then* $\Delta_2 \vdash o_2 : \tau$ *and* $\Gamma \vdash \vec{r} : \tau \rightarrowtail \sigma$ *for a* τ*.*
4. *If* $f\,o_1 = (o_2, \vec{r})$ *and* $\Delta_2 \vdash o_2 : \tau$*, then* $\Delta_1 \vdash o_1 : \sigma$ *and* $\Gamma \vdash \vec{r} : \tau \rightarrowtail \sigma$ *for a* σ*.*
5. *If* $f\,o_1 = (o_2, \vec{r})$ *and* $\Delta_1 \vdash o_1\ \mathsf{alive}$*, then* $\Delta_2 \vdash o_2\ \mathsf{alive}$*.*

The renaming function $f : \mathsf{index} \to \mathsf{option}\ (\mathsf{index} \times \mathsf{ref})$ is the core of all refinement judgments. It is used to rename object identifiers and to coalesce multiple source objects into subobjects of a single compound target object.

Consider a renaming f with $f\,o_1 = (o_1, \xhookrightarrow{\textsf{struct}\ t} 0)$ and $f\,o_2 = (o_1, \xhookrightarrow{\textsf{struct}\ t} 1)$, and an environment Γ with $\Gamma\,t = [\,\tau_1, \tau_2\,]$. This gives rise to following refinement:



Injectivity of renaming functions guarantees that distinct source objects are coalesced into disjoint target subobjects. In the case of Blazy and Leroy, the renaming functions have type $\textsf{index} \to \textsf{option}\ (\textsf{index} \times \mathbb{N})$, but we replaced the natural number by a reference since our memory model is structured using trees.

Since memory refinements rearrange the memory layout, addresses should be rearranged accordingly. The judgment $a_1 \sqsubseteq_{\Gamma}^{f:\Delta_1\mapsto\Delta_2} a_2 : \tau_{\mathsf{p}}$ describes how a_2 is obtained by renaming a_1 according to the renaming f, and moreover allows frozen union annotations $\circ$ in a_1 to be changed into unfrozen ones $\bullet$ in a_2. The index τ_{p} in the judgment $a_1 \sqsubseteq_{\Gamma}^{f:\Delta_1\mapsto\Delta_2} a_2 : \tau_{\mathsf{p}}$ corresponds to the type of a_1 and a_2.

The judgment for addresses is lifted to the judgment for pointers in the obvious way. The judgment for bits is inductively defined as:

$$\dfrac{\beta \in \{0,1\}}{\beta \sqsubseteq_{\Gamma}^{f:\Delta_1\mapsto\Delta_2} \beta} \qquad \dfrac{p_1 \sqsubseteq_{\Gamma}^{f:\Delta_1\mapsto\Delta_2} p_2 : \sigma_{\mathsf{p}} \qquad \textsf{frozen}\ p_2 \qquad i < \textsf{bitsizeof}_{\Gamma}\,(\sigma_{\mathsf{p}}*)}{(\textsf{ptr}\ p_1)_i \sqsubseteq_{\Gamma}^{f:\Delta_1\mapsto\Delta_2} (\textsf{ptr}\ p_2)_i}$$

$$\dfrac{\Gamma, \Delta_2 \vdash b}{\lightning \sqsubseteq_{\Gamma}^{f:\Delta_1\mapsto\Delta_2} b} \qquad \dfrac{\Gamma, \Delta_1 \vdash a : \sigma \qquad \Delta_1 \nvDash a\ \textsf{alive} \qquad \Gamma, \Delta_2 \vdash b}{(\textsf{ptr}\ a)_i \sqsubseteq_{\Gamma}^{f:\Delta_1\mapsto\Delta_2} b}$$

The last two rules allow indeterminate bits $\lightning$, as well as pointer fragment bits $(\textsf{ptr}\ a)_i$ belonging to deallocated storage, to be replaced by arbitrary bits b.

The judgment is lifted to memory trees following the tree structure and using the following additional rule:

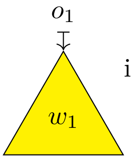
$$\dfrac{\begin{array}{c}\Gamma\,t = \vec{\tau} \qquad \Gamma, \Delta \vdash w_1 : \tau_i \qquad \overline{w_1}\,\vec{\mathbf{b}}_1 \sqsubseteq_{\Gamma}^{f:\Delta_1\mapsto\Delta_2} \vec{\mathbf{b}}_2 \qquad \vec{\mathbf{b}}_1\ \textsf{all}\ \lightning \\ \textsf{bitsizeof}_{\Gamma}\,(\textsf{union}\ t) = \textsf{bitsizeof}_{\Gamma}\,\tau_i + |\vec{\mathbf{b}}_1| \qquad \neg\textsf{unmapped}\ (\overline{w_1}\,\vec{\mathbf{b}}_1)\end{array}}{\textsf{union}_t\,(i, w_1, \vec{\mathbf{b}}_1) \sqsubseteq_{\Gamma}^{f:\Delta_1\mapsto\Delta_2} \overline{\textsf{union}_t}\,\vec{\mathbf{b}}_2 : \textsf{union}\ t}$$

This rule allows a union that has a specific variant in the source to be replaced by a union with an unspecified variant in the target. The direction seems counter intuitive, but keep in mind that unions with an unspecified variant allow more behaviors.

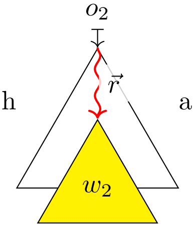
Lemma 5.8.2. *If* $w_1 \sqsubseteq_{\Gamma}^{f:\Delta_1\mapsto\Delta_2} w_2 : \tau$*, then* $\Gamma, \Delta_1 \vdash w_1 : \tau$ *and* $\Gamma, \Delta_2 \vdash w_2 : \tau$*.*

This lemma is useful because it removes the need for simultaneous inductions on both typing and refinement judgments.

We define $m_1 \sqsubseteq_\Gamma^{f} m_2$ as $m_1 \sqsubseteq_\Gamma^{f:\overline{m_1}\mapsto\overline{m_2}} m_2$, where the judgment $m_1 \sqsubseteq_\Gamma^{f:\Delta_1\mapsto\Delta_2} m_2$ is defined such that if $f\,o_1 = (o_2,\, \vec{r})$, then:



implies $\exists\, w_2\, \tau$ with



and $w_1 \sqsubseteq_\Gamma^{f:\Delta_1\mapsto\Delta_2} w_2 : \tau$.

The above definition makes sure that objects are renamed, and possibly coalesced into subobjects of a compound object, as described by the renaming function f.

In order to reason about program transformations modularly, we show that memory refinements can be composed.

Lemma 5.8.3. *Memory refinements are reflexive for valid memories, that means, if* $\Gamma, \Delta \vdash m$, *then* $m \sqsubseteq_\Gamma^{\mathsf{id}:\Delta\mapsto\Delta} m$ *where* $\mathsf{id}\, o := (o,\, \varepsilon)$.

Lemma 5.8.4. *Memory refinements compose, that means, if* $m_1 \sqsubseteq_\Gamma^{f:\Delta_1\mapsto\Delta_2} m_2$ *and* $m_2 \sqsubseteq_\Gamma^{f':\Delta_2\mapsto\Delta_3} m_3$, *then* $m_1 \sqsubseteq_\Gamma^{f'\circ f:\Delta_1\mapsto\Delta_3} m_3$ *where:*

$$(f' \circ f)\, o_1 := \begin{cases} (o_3,\, \vec{r_2}\, \vec{r_3}) & \text{if } f\, o_1 = (o_2,\, \vec{r_2}) \text{ and } f'\, o_2 = (o_3,\, \vec{r_3}) \\ \bot & \text{otherwise} \end{cases}$$

All memory operations are preserved by memory refinements. This property is not only useful for reasoning about program transformations, but also indicates that the memory interface does not expose internal details (such as variants of unions) that are unavailable in the memory of a (concrete) machine.

Lemma 5.8.5. *If* $m_1 \sqsubseteq_\Gamma^{f:\Delta_1\mapsto\Delta_2} m_2$ *and* $a_1 \sqsubseteq_\Gamma^{f:\Delta_1\mapsto\Delta_2} a_2 : \tau$ *and* $m_1\langle a_1\rangle_\Gamma = v_1$, *then there exists a value* v_2 *with* $m_2\langle a_2\rangle_\Gamma = v_2$ *and* $v_1 \sqsubseteq_\Gamma^{f:\Delta_1\mapsto\Delta_2} v_2 : \tau$.

Lemma 5.8.6. *If* $m_1 \sqsubseteq_\Gamma^{f:\Delta_1\mapsto\Delta_2} m_2$ *and* $a_1 \sqsubseteq_\Gamma^{f:\Delta_1\mapsto\Delta_2} a_2 : \tau$ *and* $v_1 \sqsubseteq_\Gamma^{f:\Delta_1\mapsto\Delta_2} v_2 : \tau$ *and* $\mathsf{writable}_\Gamma\, m_1\, a_1$, *then:*

1. *We have* $\mathsf{writable}_\Gamma\, m_2\, a_2$.
2. *We have* $m_1\langle a_1 := v_1\rangle_\Gamma \sqsubseteq_\Gamma^{f:\Delta_1\mapsto\Delta_2} m_2\langle a_2 := v_2\rangle_\Gamma$.

As shown in Lemma 5.6.10, storing a value v in memory and then retrieving it, does not necessarily yield the same value v. In case of a byte address, the value may have been stored in padding and therefore have become indeterminate. Secondly, it intentionally yields the value $|\, v\,|_\circ$ in which all pointers are frozen. However, the widely used compiler optimization of constant propagation, which substitutes values of known constants at compile time, is still valid in our memory model.

Lemma 5.8.7. *If* $\Gamma, \Delta \vdash v : \tau$, *then* $|\, v\,|_\circ \sqsubseteq_\Gamma^{\Delta} v : \tau$.

Theorem 5.8.8 (Constant propagation)**.** *If* $\Gamma, \Delta \vdash m$ *and* $\Gamma, \Delta \vdash a : \tau$ *and* $\Gamma, \Delta \vdash v : \tau$ *and* $\mathsf{writable}_\Gamma\, a\, m$*, then there exists a value* v' *with:*

$$m\langle a := v\rangle_\Gamma \langle a\rangle_\Gamma = v' \quad \text{and} \quad v' \sqsubseteq_\Gamma^\Delta v : \tau.$$

Copying an object w by an assignment results in it being converted to a value $\mathsf{toval}_\Gamma\, w$ and back. This conversion makes invalid representations of base values indeterminate. Copying an object w byte-wise results in it being converted to bits $\overline{w}$ and back. This conversion makes all variants of unions unspecified. The following theorem shows that a copy by assignment can be transformed into a byte-wise copy.

Theorem 5.8.9 (Memcpy)**.** *If* $\Gamma, \Delta \vdash w : \tau$*, then:*

$$\mathsf{ofval}_\Gamma\, (\overline{w}_{\mathbf{1}})\, (\mathsf{toval}_\Gamma\, w) \sqsubseteq_\Gamma^\Delta w \sqsubseteq_\Gamma^\Delta (\overline{w})_\Gamma^\tau : \tau.$$

Unused reads cannot be removed unconditionally in the CH$_2$O memory model because these have side-effects in the form of uses of the force_Γ operation that updates effective types. We show that uses of force_Γ can be removed for frozen addresses.

Theorem 5.8.10. *If* $\Gamma, \Delta \vdash m$ *and* $m\langle a\rangle_\Gamma \neq \bot$ *and* $\mathsf{frozen}\ a$*, then* $\mathsf{force}_\Gamma\, a\, m \sqsubseteq_\Gamma^\Delta m$*.*

Chapter 6

CH_2O core C

This chapter describes the language CH_2O core C. The syntax of this language resembles actual C but incorporates many simplifications to make its semantics more principled. For example, CH_2O core C has just one looping construct that generalizes the various looping constructs of C (**`while`**, **`for`** and **`do-while`** loops), uses De Bruijn indices for local variables and makes l-value conversion explicit. Chapter 7 shows how C abstract syntax is translated into CH_2O core C.

This chapter defines three versions of the semantics of CH_2O core C. A small-step operational semantics, an executable semantics and an evaluator for pure expressions. We summarize the key points of these semantics below.

CH_2O core C is typed, which means that each syntactical category of the language has a typing judgment. We establish desirable properties such as type preservation (reduction in the operational semantics preserves typing, see Theorem 6.6.13), progress (every non-final non-error state can reduce, see Theorem 6.6.14) and invariance under memory refinements (Theorem 6.7.1).

Operational semantics. Computation in the small-step operational semantics is defined as the reflexive transitive closure of a reduction relation $\Gamma, \delta \vdash S_1 \rightarrow S_2$. The environment $\delta \in \mathsf{funenv}$ contains the bodies of all declared functions.

States $S \in \mathsf{state}$ consist of two components: the memory and the current location in the program that is being executed. The current location in the program is described using an adaptation of Huet's zipper data structure [Hue97] over statement abstract syntax trees. We use the zipper data structure to accurately describe non-local control (**`goto`**, **`break`**, **`continue`**, **`return`** and unstructured **`switch`**) in the presence of block scope local variables. The current location in the program is a pair $(\mathcal{P}, \phi)$ where $\mathcal{P} \in \mathsf{ctx}$ is the context of the part $\phi \in \mathsf{focus}$ that is being executed. We consider the following forms of program execution:

- **Execution of a statement.** Execution of a statement occurs by small-step traversal through the zipper in a direction corresponding to the current form of (non-local) control. The context $\mathcal{P}$ of our zipper implicitly contains the program stack $\rho \in \mathsf{stack}$ that assigns locations in memory to local variables.

- **Execution of an expression.** Execution of an expression is described by a head reduction $\Gamma, \rho \vdash (e_1, m_1) \multimap_{\mathsf{h}} (e_2, m_2)$. Evaluation contexts [FFKD87] are used to select a head redex in a whole expression.
- **Calling a function** and **returning from a function.** When calling a function, the zipper is extended with the function body of the callee, which is in turn executed. Returning from a function resumes execution of the caller, whose location is stored as part of the context $\mathcal{P}$ of our zipper.
- **Undefined behavior.** We use a special $\overline{\mathsf{undef}}$ state to describe that a run-time error (undefined behavior in C terminology) has occurred.

Executable semantics. The executable semantics $S_2 \in \mathsf{exec}_{\Gamma,\delta}\ S_1$ is an algorithmic version of the operational semantics. That means, the function $\mathsf{exec}_{\Gamma,\delta} : \mathsf{state} \to \mathcal{P}_{\mathsf{fin}}(\mathsf{state})$ *computes* the finite set of subsequent states.

The executable semantics is proven sound (Theorem 6.8.2) and complete (Theorem 6.8.7) with respect to the operational semantics.

Pure expression evaluation. Pure expressions do not contain assignments and function calls and therefore enjoy three useful properties: they do not modify the memory state, they yield unique results, and their executions always terminate. These properties allow us to define an evaluator $[\![\,\_\,]\!]_{\Gamma,\rho,m} : \mathsf{expr} \to \mathsf{option}\ \mathsf{lrval}$ that yields the resulting address or abstract value of an expression.

The evaluator for pure expressions is proven sound (Theorem 6.9.5) and complete (Theorem 6.9.7) with respect to the operational semantics.

6.1 Expressions

This section defines the syntax of expressions of CH$_2$O core C. Although the semantics will only be given in Sections 6.3 and 6.4, we already explain the informal meaning of the different expression constructs and define their typing rules.

The C language distinguishes expressions that are *l-values* and *r-values* [ISO12, 6.3.2.1]. This distinction originates from the assignment `e1 = e2` where the left hand side `e1` does not designate a value, but a pointer that refers to a value in memory. In CH$_2$O core C we let l-values reduce to pointers and r-values to abstract values. L-values reduce to pointers rather than addresses because they can be `NULL` [ISO12, 6.5.3.2p3]. L-values of function type account for function designators [ISO12, 6.3.2.1].

Definition 6.1.1. Left-right values *are defined as:*

$$\nu \in \mathsf{lrval} := \mathsf{ptr} + \mathsf{val}.$$

The C language supports simple assignments `e1 = e2` [ISO12, 6.5.16.1], compound assignments [ISO12, 6.5.16.1] such as `e1 += e2`, and increment and decrement operators [ISO12, 6.5.3.1, 6.5.2.4] such as `e++` and `--e`. CH$_2$O core C generalizes these forms of assignments.

Definition 6.1.2. Assignment kinds *are inductively defined as:*

$$\alpha \in \mathsf{assign} ::= \ := \ \mid \circledcirc{:=} \ \mid \ {:=}\circledcirc.$$

The syntax of binary operators $\circledcirc$ *is given in Definition 3.2.1 on page 41.*

Definition 6.1.3. *The judgment* $\tau_1 \ \alpha \ \tau_2 : \sigma$ *describes the* resulting type σ of an assignment α with operands of types τ_1 and τ_2. *It is inductively defined as:*

$$\frac{(\tau_1)\tau_2 : \tau_1}{\tau_1 := \tau_2 : \tau_1} \qquad \frac{\tau_1 \circledcirc \tau_2 : \sigma \qquad (\tau_1)\sigma : \tau_1}{\tau_1 \circledcirc{:=} \tau_2 : \tau_1} \qquad \frac{\tau_1 \circledcirc \tau_2 : \sigma \qquad (\tau_1)\sigma : \tau_1}{\tau_1 :=\circledcirc \tau_2 : \tau_1}$$

The judgments $\circledcirc_u \tau : \sigma$, $\tau_1 \circledcirc \tau_2 : \sigma$, and $(\sigma)\tau : \sigma$ describe typing of unary, binary and cast operators. They are defined following the structure of types.

We now give an informal description of the semantics of assignments:

- The assignment $e_1 := e_2$ performs a simple assignment. It assigns the value of e_2 to the pointer designated by e_1. It yields the value of e_2.
- The assignment $e_1 \circledcirc{:=} e_2$ performs a compound assignment. It assigns the value of $e_1 \circledcirc e_2$ to the pointer designated by e_1. It yields the value of $e_1 \circledcirc e_2$.
- The assignment $e_1 :=\circledcirc e_2$ generalizes the postfix increment and postfix decrement operators. It assigns the value of $e_1 \circledcirc e_2$ to the pointer designated by e_1. It yields the original value of e_1.

The above description is subject to the implicit type conversions shown in Definition 6.1.3. The formal semantics of assignments is given in Definition 6.3.7.

Definition 6.1.4. CH$_2$O core C expressions *are inductively defined as:*

$$\begin{aligned}
e \in \mathsf{expr} ::= {} & x_i \mid [\nu]_\Omega && \text{(variables and constants)}\\
\mid {} & *e \mid \&e && \text{(l-value and r-value conversion)}\\
\mid {} & e \ \mathbf{.}_{\mathbf{l}} \ r \mid e \ \mathbf{.}_{\mathbf{r}} \ r && \text{(indexing of arrays, structs and unions)}\\
\mid {} & e_1[\vec{r} := e_2] && \text{(altering arrays, structs and unions)}\\
\mid {} & e_1 \ \alpha \ e_2 \mid \mathsf{load}\ e && \text{(assignments and loading from memory)}\\
\mid {} & e(\vec{e}) \mid \mathsf{abort}\ \tau && \text{(function calls and undefined behavior)}\\
\mid {} & \mathsf{alloc}_\tau\ e \mid \mathsf{free}\ e && \text{(allocation and deallocation)}\\
\mid {} & \circledcirc_u\ e \mid e_1 \circledcirc e_2 \mid (\tau)e && \text{(unary, binary and cast operators)}\\
\mid {} & (e_1, e_2) \mid e_1 \ ? \ e_2 : e_3 && \text{(comma and sequenced conditional)}
\end{aligned}$$

The syntax of unary operators $\circledcirc_u$ *and binary operators* $\circledcirc$ *is given in Definition 3.2.1 on page 41.*

Notation 6.1.5. *We often abbreviate* $[\nu]_\emptyset$ *as* ν.

Notation 6.1.6. *We let* $[\vec{\nu}]_{\vec{\Omega}}$ *denote* $[\nu_0]_{\Omega_0}, \ldots, [\nu_{n-1}]_{\Omega_{n-1}}$ *where* $n = |\vec{\Omega}| = |\vec{\nu}|$.

We define typing judgments $\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{l}} e : \tau_{\mathsf{p}}$ for l-values and $\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e : \tau$ for r-values. The list $\vec{\tau} \in \mathsf{list\ type}$ gives the types of local variables, which are represented as De Bruijn indices [Bru72]. This means that a local variable x_i refers to the i-th enclosing local variable declaration (Definition 6.2.2), whose type is τ_i.

Definition 6.1.7. *The judgment* $\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{l}} e : \tau_{\mathsf{p}}$ *describes that* e is an l-expression of type τ*, and* $\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e : \tau$ *describes that* e is an r-expression of type τ*. These judgments are mutually inductively defined as:*

$$\frac{}{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{l}} x_i : \tau_i} \qquad \frac{\Gamma, \Delta \vdash \Omega \qquad \Gamma, \Delta \vdash v : \tau}{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} [v]_{\Omega} : \tau} \qquad \frac{\Gamma, \Delta \vdash \Omega \qquad \Gamma, \Delta \vdash a : \tau_{\mathsf{p}}}{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{l}} [a]_{\Omega} : \tau_{\mathsf{p}}}$$

$$\frac{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e : \tau_{\mathsf{p}}*}{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{l}} *e : \tau_{\mathsf{p}}} \qquad \frac{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{l}} e : \tau_{\mathsf{p}}}{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} \&e : \tau_{\mathsf{p}}*}$$

$$\frac{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{l}} e : \tau \qquad \Gamma \vdash r : \tau \rightarrowtail \sigma}{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{l}} e \;._{\mathbf{l}}\; r : \sigma} \qquad \frac{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e : \tau \qquad \Gamma \vdash r : \tau \rightarrowtail \sigma}{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e \;._{\mathbf{r}}\; r : \sigma}$$

$$\frac{\Gamma \vdash \vec{r} : \tau \rightarrowtail \sigma \qquad \Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e_1 : \tau \qquad \Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e_2 : \sigma}{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e_1[\vec{r} := e_2] : \tau}$$

$$\frac{\tau_1 \;\alpha\; \tau_2 : \sigma \qquad \Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{l}} e_1 : \tau_1 \qquad \Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e_2 : \tau_2}{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e_1 \;\alpha\; e_2 : \sigma} \qquad \frac{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{l}} e : \tau \qquad \mathsf{complete}_{\Gamma}\; \tau}{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} \mathsf{load}\; e : \tau}$$

$$\frac{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e : \vec{\sigma} \to \tau \qquad \Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} \vec{e} : \vec{\sigma} \qquad \mathsf{complete}_{\Gamma}\; \tau}{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e(\vec{e}) : \tau} \qquad \frac{\Gamma \vdash \tau}{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} \mathsf{abort}\; \tau : \tau}$$

$$\frac{\Gamma \vdash \tau \qquad \Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e : \sigma_{\mathsf{i}}}{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{l}} \mathsf{alloc}_{\tau}\; e : \tau} \qquad \frac{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{l}} e : \tau_{\mathsf{p}}}{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} \mathsf{free}\; e : \mathsf{void}}$$

$$\frac{\circledcirc_u \tau : \sigma \qquad \Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e : \tau}{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} \circledcirc_u e : \sigma} \qquad \frac{\tau_1 \circledcirc \tau_2 : \sigma \qquad \Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e_1 : \tau_1 \qquad \Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e_2 : \tau_2}{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e_1 \circledcirc e_2 : \sigma}$$

$$\frac{(\sigma)\tau : \sigma \qquad \Gamma \vdash \sigma \qquad \Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e : \tau}{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} (\sigma)e : \sigma} \qquad \frac{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e_1 : \tau_1 \qquad \Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e_2 : \tau_2}{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} (e_1, e_2) : \tau_2}$$

$$\frac{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e_1 : \tau_{\mathsf{b}} \qquad \tau_{\mathsf{b}} \neq \mathsf{void} \qquad \Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e_2 : \sigma \qquad \Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e_3 : \sigma}{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e_1 \;?\; e_2 : e_3 : \sigma}$$

Notation 6.1.8. *We sometimes combine typing of l-expressions and r-expressions using the shorthand* $\Gamma, \Delta, \vec{\tau} \vdash e : \tau_{\mathsf{lr}}$ *where* $\tau_{\mathsf{lr}} \in \mathsf{ptrtype} + \mathsf{type}$.

In actual C, l-values that do not have array type are implicitly converted to r-values by *l-value conversion* [ISO12, 6.3.2.1p2], and l-values of array type are implicitly converted to r-values of pointer type that refer to the first element of the array [ISO12, 6.3.2.1p3]. In CH$_2$O core C we make these conversions explicit using the expressions $\&e$, $*e$ and $\mathsf{load}\; e$ that are meant to map between l-values and r-values. The expressions $\&e$ is the coercion from l-values into r-values, and $*e$ is its partial inverse. The expression $\mathsf{load}\; e$ is used to obtain the stored value designated by an l-value e.

Due to explicit l-value conversion, we have operators to select fields of structs and unions as l-values $e \;._{\mathbf{l}}\; r$ and as r-values $e \;._{\mathbf{r}}\; r$. The semantics of these operators is entirely different. The operator $e \;._{\mathbf{l}}\; r$ operates on pointers whereas $e \;._{\mathbf{r}}\; r$ operates on values. Let us illustrate the difference:

```
struct S { int x; int y; } s;
struct S f() { struct S r = { 2, 3 }; return r; }
```

The expression `s.x` performs field indexing on an l-value because `s` is an l-value, whereas `f().x` performs field indexing on an r-values because `f()` is an r-value.

The operator $e \;._{\mathsf{l}}\; r$ is also used to convert l-values of array type into r-values of pointer type that refer to the first element of the array. Consider:

```
int a[4], *p = a;
```

The right hand side of the assignment `*p = a` corresponds to the following CH_2O core C expression:

$$\&\Big(x_{\mathsf{a}} \;._{\mathsf{l}}\; (\xhookrightarrow{\mathsf{signed\ int}[4]} 0)\Big).$$

The leaves of expressions $[\nu]_{\Omega}$ are annotated with a finite set $\Omega \in \mathsf{lockset}$ of *locked* addresses. The set Ω is used to describe the sequence point restriction. Whenever an assignment is performed, the affected object is locked in memory and its address is added to Ω. Locking an address enforces the sequence point restriction because permissions enforce that consecutive accesses to that address will fail. At the subsequent sequence point, the objects of the addresses in Ω are unlocked in order to allow future accesses to the assigned objects. Section 6.4 formally treats the sequence point restriction as part of the operational semantics.

This set is only used for bookkeeping during program execution. It is initially empty as ensured by the typing judgment for statements (see Definition 6.2.4).

Definition 6.1.9. *The function* $\mathsf{locks} : \mathsf{expr} \to \mathsf{lockset}$ *collects the set of lock annotations of an expression.*

Definition 6.1.10. Stacks *are defined as:*

$$\rho \in \mathsf{stack} := \mathsf{list}\ (\mathsf{index} \times \mathsf{type}).$$

Definition 6.1.11. *The judgment* $\Delta \vdash \rho$ *describes that* ρ is a valid stack in Δ. *The judgment holds iff* $\Delta \vdash o : \tau$ *for each* $(o, \tau) \in \rho$. *Validity of memory indices* $\Delta \vdash o : \tau$ *has been defined in Definition 5.2.2 on page 64.*

Accessing the value of a local variable has an additional level of indirection. A local variable x_i represented as a De Bruijn index refers to the i-th item on the stack ρ, which contains a reference to the value in memory instead of the value of the variable itself. This way, pointers to both local and allocated storage are treated uniformly. Evaluation of x_i consists of looking up its object identifier o in the stack ρ, and yields an address to the memory tree at position o in memory. The value of a variable can be obtained through a load.

The first operand e of a function call $e(\vec{e})$ is an expression instead of a function name. This expression should yield a function pointer to the callee.

The expression $\mathsf{abort}\ \tau$ results in undefined behavior when evaluated. The translation from CH_2O abstract C inserts $\mathsf{abort}\ \tau$ expressions in branches of the program that have undefined behavior when reached.

The expressions $\mathsf{alloc}_{\tau}\ e$ and $\mathsf{free}\ e$ are used for allocation and deallocation of dynamically allocated memory. These correspond to the C11 standard library functions `malloc` [ISO12, 7.22.3.4] and `free` [ISO12, 7.22.3.3], but we treat them as primitives.

Note that because of effective types `malloc` and `free` cannot be implemented in terms of standards compliant C code. We currently only supports invocations of `malloc` of the following shape:

```
malloc(e * sizeof(τ))
```

Our $\mathsf{alloc}_\tau\ e$ expression corresponds to invocations of `malloc` as these. Its type is $\tau *$ and it is thus similar to the type-safe `new` operator of C++. An untyped version of `malloc` that allocates a chunk of untyped memory is left for future work.

The expression $e_1[\vec{r} := e_2]$ is used to encode initializers [ISO12, 6.7.9] and compound literals [ISO12, 6.5.2.5]. Compound literals are expressions that denote struct, union and array values. For example:

```
struct S { int x, y[2], z; };
(struct S){ .y[1] = f(), g(), .x = 10 };
```

Compound literals are already complicated from a syntactical point of view: subobjects can be initialized hereditarily (the initializer `.y[1]=f()` specifies the second element of `y`), initializers may occur in any order, and field names (also called designators) may be omitted. If the designator is omitted, the next object in the tree structure is initialized (in the example, `g()` initializes the field `z`). From a semantical point of view, side-effects may be executed in any order [ISO12, 6.7.9p23].

During the translation from CH$_2$O abstract C into CH$_2$O core C, we translate each compound literal into a sequence of $\_[\_ := \_]$ constructs. The CH$_2$O core C expression corresponding to the above compound literal is:

$$\begin{aligned}
\mathbf{0}_\Gamma^{\mathsf{struct\ S}}\ &[(\xhookrightarrow{\mathsf{struct\ S}} 1 \xhookrightarrow{\mathsf{signed\ int}[1]} 0) := (\mathsf{ptr}\,\mathsf{f}^{\mathsf{void} \mapsto \mathsf{signed\ int}})(\varepsilon)] \\
&[(\xhookrightarrow{\mathsf{struct\ S}} 2) := (\mathsf{ptr}\,\mathsf{g}^{\mathsf{void} \mapsto \mathsf{signed\ int}})(\varepsilon)] \\
&[(\xhookrightarrow{\mathsf{struct\ S}} 0) := \mathsf{int}_{\mathsf{signed\ int}}\ 10]
\end{aligned}$$

The value $\mathbf{0}_\Gamma^{\mathsf{struct\ S}}$ is a struct initialized with zeros (see Definition 6.3.12). Fields that are not explicitly initialized are implicitly initialized with the zero value [ISO12, 6.7.9p10]. Our non-deterministic semantics for expressions allows the function calls to `f` and `g` to be executed in any order.

6.2 Statements

This section defines the syntax and typing rules of the statements of CH$_2$O core C. We have omitted the (unstructured) `switch` statement for brevity's sake but it is part of the Coq formalization.

Definition 6.2.1. Label names $l \in \mathsf{labelname}$ *are represented as strings.*

Definition 6.2.2. CH$_2$O core C statements *are inductively defined as:*

$$\begin{array}{rll}
s \in \mathsf{stmt} ::= & e \mid \mathsf{return}\ e & \text{(expression and } \mathtt{return} \text{ statements)}\\
 & \mid \mathsf{goto}\ l \mid l : & (\mathtt{goto} \text{ and label})\\
 & \mid \mathsf{throw}\ n \mid \mathsf{catch}\ s & (\mathtt{throw} \text{ and } \mathtt{catch})\\
 & \mid \mathsf{skip} \mid s_1\ ;\ s_2 & \text{(the skip statement and composition)}\\
 & \mid \mathsf{local}_\tau\ s & \text{(block scope local variable declaration)}\\
 & \mid \mathsf{loop}\ s & \text{(infinite loop)}\\
 & \mid \mathsf{if}\ (e)\ s_1\ \mathsf{else}\ s_2 & \text{(conditional statement)}
\end{array}$$

The typing judgment for statements is of the shape $\Gamma, \Delta, \vec{\tau} \vdash s : (\beta,\ \tau^?)$. The type of the return statements is given by $\tau^? \in \mathsf{option\ type}$, which is $\bot$ if s does not contain any returns. The Boolean β denotes whether s is statically known to always return.

Definition 6.2.3. *The judgment* $\tau_1^?;\tau_2^? \succ \tau^?$ *denotes that* $\tau^?$ *is the* combined return type of $\tau_1^?$ and $\tau_2^?$. *It is inductively defined as:*

$$\frac{}{\tau^?;\tau^? \succ \tau^?} \qquad \frac{}{\bot;\tau \succ \tau} \qquad \frac{}{\tau;\bot \succ \tau}$$

Definition 6.2.4. *The judgment* $\Gamma, \Delta, \vec{\tau} \vdash s : (\beta,\ \tau^?)$ *describes that* the statement s has return type $\tau^?$ and is known to always return if $\beta = \mathsf{true}$. *It is inductively defined as:*

$$\frac{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e : \tau \qquad \mathsf{locks}\ e = \emptyset}{\Gamma, \Delta, \vec{\tau} \vdash e : (\mathsf{false},\ \bot)} \qquad \frac{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e : \tau \qquad \mathsf{locks}\ e = \emptyset}{\Gamma, \Delta, \vec{\tau} \vdash \mathsf{return}\ e : (\mathsf{true},\ \tau)}$$

$$\frac{}{\Gamma, \Delta, \vec{\tau} \vdash \mathsf{goto}\ l : (\mathsf{true},\ \bot)} \qquad \frac{}{\Gamma, \Delta, \vec{\tau} \vdash (l :) : (\mathsf{false},\ \bot)}$$

$$\frac{}{\Gamma, \Delta, \vec{\tau} \vdash \mathsf{throw}\ n : (\mathsf{true},\ \bot)} \qquad \frac{\Gamma, \Delta, \vec{\tau} \vdash s : (\beta,\ \sigma^?)}{\Gamma, \Delta, \vec{\tau} \vdash \mathsf{catch}\ s : (\mathsf{false},\ \sigma^?)}$$

$$\frac{}{\Gamma, \Delta, \vec{\tau} \vdash \mathsf{skip} : (\mathsf{false},\ \bot)} \qquad \frac{\Gamma, \Delta, \vec{\tau} \vdash s_1 : (\beta_1,\tau_1^?) \quad \Gamma, \Delta, \vec{\tau} \vdash s_2 : (\beta_2,\tau_2^?) \quad \tau_1^?;\tau_2^? \succ \sigma^?}{\Gamma, \Delta, \vec{\tau} \vdash s_1\ ;\ s_2 : (\beta_2,\ \sigma^?)}$$

$$\frac{\Gamma \vdash \tau \qquad \Gamma, \Delta, \tau\,\vec{\tau} \vdash s : (\beta,\ \sigma^?)}{\Gamma, \Delta, \vec{\tau} \vdash \mathsf{local}_\tau\ s : (\beta,\ \sigma^?)} \qquad \frac{\Gamma, \Delta, \vec{\tau} \vdash s : (\beta,\ \sigma^?)}{\Gamma, \Delta, \vec{\tau} \vdash \mathsf{loop}\ s : (\mathsf{true},\ \sigma^?)}$$

$$\frac{\begin{array}{c}\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e : \tau_{\mathsf{b}} \qquad \tau_{\mathsf{b}} \neq \mathsf{void} \qquad \mathsf{locks}\ e = \emptyset\\ \Gamma, \Delta, \vec{\tau} \vdash s_1 : (\beta_1,\ \tau_1^?) \qquad \Gamma, \Delta, \vec{\tau} \vdash s_2 : (\beta_2,\ \tau_2^?) \qquad \tau_1^?;\tau_2^? \succ \sigma^?\end{array}}{\Gamma, \Delta, \vec{\tau} \vdash \mathsf{if}\ (e)\ s_1\ \mathsf{else}\ s_2 : (\beta_1 \wedge \beta_2,\ \sigma^?)}$$

The statement $\mathsf{local}_\tau\ s$ opens a block scope with one local variable of type τ. Since we use De Bruijn indices, the statement $\mathsf{local}_\tau\ s$ is nameless.

The C language supports **`while`**, **`for`** and **`do-while`** looping statements [ISO12, 6.8.5]. We make the semantics of CH$_2$O core C more principled by factoring out similarities in these looping statements.

We adapt the treatment of loops by Leroy [Ler06] and use a construct $\mathsf{loop}\ s$ for an infinite loop, which combined with the $\mathsf{throw}\ n$ statement that jumps to the nth surrounding catch construct, can encode all C loops:

$$\begin{aligned}
\mathsf{while}(e)\ s &\Rightarrow \mathsf{catch}\ (\mathsf{loop}\ (\mathsf{if}\ (e)\ \mathsf{skip}\ \mathsf{else}\ \mathsf{throw}\ 0\,;\ \mathsf{catch}\ s))\\
\mathsf{for}(e_1\ ;\ e_2\ ;\ e_3)\ s &\Rightarrow e_1\ ;\ \mathsf{catch}\ (\mathsf{loop}\ (\mathsf{if}\ (e_2)\ \mathsf{skip}\ \mathsf{else}\ \mathsf{throw}\ 0\,;\ \mathsf{catch}\ s\,;\ e_3))\\
\mathsf{do}\ s\ \mathsf{while}(e) &\Rightarrow \mathsf{catch}\ (\mathsf{loop}\ (\mathsf{catch}\ s\,;\ \mathsf{if}\ (e)\ \mathsf{skip}\ \mathsf{else}\ \mathsf{throw}\ 0))
\end{aligned}$$

In this setting, break statements are encoded as $\mathsf{throw}\ 1$ statements and $\mathsf{continue}$ statements are encoded as $\mathsf{throw}\ 0$ statements.

One could encode throw and catch using `goto`s. We have tried this approach, but it turned out to be inconvenient due to freshness conditions on labels.

Definition 6.2.5. CH$_2$O core C programs *are defined as:*

$$\delta \in \mathsf{funenv} := \mathsf{funname} \to_{\mathsf{fin}} \mathsf{stmt}.$$

The judgment $\Gamma, \Delta \vdash \delta$ for programs ensures that each function body in δ is well-typed with respect to its corresponding prototype in Γ and ensures that all `goto`s and `throw`s have a corresponding labeled statement or `catch`. We will first introduce some auxiliary definitions to express these properties formally.

Definition 6.2.6. *The function* $\mathsf{labels} : \mathsf{stmt} \to \mathcal{P}_{\mathsf{fin}}(\mathsf{labelname})$ *collects the set of labels of labeled substatements. The function* $\mathsf{gotos} : \mathsf{stmt} \to \mathcal{P}_{\mathsf{fin}}(\mathsf{labelname})$ *collects the set of labels of* `goto`*s.*

Definition 6.2.7. *The judgment $s \uparrow n$ describes that* all `throw`s in s will be caught if s is surrounded by n level of `catch`es. *It is inductively defined as:*

$$\frac{}{e \uparrow n} \qquad \frac{}{\mathsf{return}\ e \uparrow n} \qquad \frac{}{\mathsf{goto}\ l \uparrow n} \qquad \frac{}{(l :) \uparrow n} \qquad \frac{i < n}{\mathsf{throw}\ i \uparrow n} \qquad \frac{s \uparrow n+1}{\mathsf{catch}\ s \uparrow n}$$

$$\frac{}{\mathsf{skip} \uparrow n} \qquad \frac{s_1 \uparrow n \qquad s_2 \uparrow n}{s_1\ ;\ s_2 \uparrow n} \qquad \frac{s \uparrow n}{\mathsf{local}_\tau\ s \uparrow n} \qquad \frac{s \uparrow n}{\mathsf{loop}\ s \uparrow n} \qquad \frac{s_1 \uparrow n \qquad s_2 \uparrow n}{\mathsf{if}\ (e)\ s_1\ \mathsf{else}\ s_2 \uparrow n}$$

Definition 6.2.8. *The judgment $(\beta,\ \tau^?) \succ \sigma$ denotes that σ* is a valid return type of $(\beta,\ \tau^?)$. *It is inductively defined as:*

$$\frac{}{(\mathsf{true},\ \sigma) \succ \sigma} \qquad \frac{}{(\mathsf{false},\ \mathsf{void}) \succ \mathsf{void}} \qquad \frac{}{(\mathsf{true},\ \bot) \succ \sigma} \qquad \frac{}{(\mathsf{false},\ \bot) \succ \mathsf{void}}$$

Definition 6.2.9. *The judgment $\Gamma, \Delta \vdash \delta$ describes that* a program δ is valid. *For each function name f and statement s with $\delta\, f = s$ there should be argument types $\vec{\tau}$ and a return type σ with $\Gamma\, f = (\vec{\tau},\ \sigma)$ and:*

1. *The function body is well-typed:* $\Gamma, \Delta, \vec{\tau} \vdash s : (\beta,\ \tau^?)$.
2. *The return type matches the prototype:* $(\beta,\ \tau^?) \succ \sigma$.
3. *The argument types are complete:* $\mathsf{complete}_\Gamma\ \vec{\tau}$ *(see Definition 3.3.7 on page 47).*
4. *All* `goto`*s have a corresponding labeled statement:* $\mathsf{gotos}\ s \subseteq \mathsf{labels}\ s$.

5. *All* `throws` *have a corresponding* `catch`*:* $s \uparrow 0$*.*

Furthermore, all prototypes should have a declaration: $\mathsf{dom}_{\mathsf{funname}}\, \Gamma \subseteq \mathsf{dom}\ \delta$*.*

The judgment for programs requires each branch of a non-void function to have a return statement. This is not the case in:

```
int f() { while(1) { /* do something */ } }
```

A return ($\mathsf{abort\ signed\ int}$) statement will be inserted by the translation from CH_2O abstract C into CH_2O core C. This approach captures all undefined behaviors due to control reaching the end of a non-void function [ISO12, 6.9.1p12].

6.3 Semantics of operators

This section describes the semantics of the expression constructs for binary operators $e_1 \circledcirc e_2$, unary operators $\circledcirc_u\, e$, casts $(\tau)e$, assignments $e_1\ \alpha\ e_2$, indexing of compound types $r\ ._{\mathbf{l}}\ e$ and $r\ ._{\mathbf{r}}\ e$, and altering of compound types $e_1[\vec{r} := e_2]$.

6.3.1 Pointer comparisons

As shown in Section 2.5.8, end-of-array pointers give rise to subtle corner cases:

```
int x, y;
if (&x + 1 == &y) printf("x and y are allocated adjacently\n");
```

According to the C11 [ISO12, 6.5.9p6] standard, the `printf` is executed depending on the way `&x` and `&y` are allocated in the address space. However, in GCC this is not always the case, see the example in Section 2.5.8. Due to discrepancies between the C11 standard [ISO12, 6.5.9p6] and Defect Report #260 [ISO] it is unclear if this is a bug in GCC. We thus assign undefined behavior to these questionable comparisons involving end-of-array pointers.

Definition 6.3.1. *The judgment* $\Gamma, m \vdash (a_1 \circledcirc_c a_2)\ \mathsf{defined}$ *describes if* comparison of addresses a_1 and a_2 is defined*. It is inductively defined as:*

$$\frac{\overline{m} \vdash \{a_1, a_2\}\ \mathsf{alive} \qquad \mathsf{index}\ a_1 \neq \mathsf{index}\ a_2 \to (\circledcirc_c = (\mathtt{==})\ \textit{and}\ \Gamma \vdash \{a_1, a_2\}\ \mathsf{strict})}{\Gamma, m \vdash (a_1 \circledcirc_c a_2)\ \mathsf{defined}}$$

Comparison operators $\circledcirc_c \in \mathsf{compop}$ *are defined in Definition 3.2.1 on page 41. The* result of the comparison $(a_1 \circledcirc_c a_2)$ *is defined as follows:*

$$(a_1 \circledcirc_c a_2) := \mathsf{index}\ a_1 = \mathsf{index}\ a_2 \wedge \mathsf{bitoffset}_\Gamma\ a_1 \circledcirc_c \mathsf{bitoffset}_\Gamma\ a_2.$$

The bit-offset $\mathsf{bitoffset}_\Gamma\ a$ *of an address* a *is defined in Definition 5.2.15 on page 70.*

The judgment $\Gamma, m \vdash (a_1 \circledcirc_c a_2)\ \mathsf{defined}$ distinguishes two cases:

- Both addresses point into the same object ($\mathsf{index}\ a_1 = \mathsf{index}\ a_2$). In this case it does not matter if one of the addresses is end-of-array.

- The addresses point into different objects ($\mathsf{index}\ a_1 \neq \mathsf{index}\ a_2$). In this case both should be strict (that is, not end-of-array, see Definition 5.2.8 on page 68) and one may only test for equality ==. Inequality comparisons < or <= of addresses that point into different objects have undefined behavior [ISO12, 6.5.8p5].

Comparison ignores the tree structure of the surrounding object entirely since it is defined in terms of bit-offsets $\mathsf{bitoffset}_\Gamma$. Let us consider some examples:

```
struct S { int a[3]; int b[3]; } s1, s2;
s1.a == s2.b;   // OK, neither of the two pointers is end-of-array
s1.a == s1.b+3; // OK, same object
s1.a == s2.b+3; // Undefined, different objects, s2.b+3 end-of-array
s1.a <= s1.b;   // OK, <= into the same object
s1.a <= s2.a;   // Undefined, <= with different objects
```

It is important to note that the common programming practice of using end-of-array pointer comparisons in loops through arrays is allowed in CH$_2$O. For example, the program from Section 2.5.8 that increases all values of a has defined behavior:

```
int a[4] = { 0, 1, 2, 3 };
int *p = a; // p and a point into the same object
while (p < a + 4) { *p += 1; p += 1; }
```

As shown in Section 2.5.7, using an indeterminate pointer in a pointer comparison has undefined behavior. The side-condition $\overline{m} \vdash \{a_1, a_2\}\ \mathsf{alive}$ of the judgment $\Gamma, m \vdash (a_1 \circledcirc_c a_2)\ \mathsf{defined}$ ensures that the given addresses have not been deallocated (see also Definition 5.2.2 on page 64). For example:

```
int *p = malloc(sizeof(int)); assert (p != NULL);
free(p);
int *q = malloc(sizeof(int)); assert (q != NULL);
if (p == q) { ... } // Undefined, p refers to deallocated memory
```

6.3.2 Pointer subtraction

The C11 standard puts stronger requirements on pointer subtraction than on pointer comparison. Pointer subtraction has defined behavior only if the given pointers both point to elements of the *same array object* [ISO12, 6.5.6p9]. This is a stronger requirement than that both pointers should be part of the same memory object as used for pointer comparisons. For example:

```
struct S { int a[3]; int b[3]; } s;
s.a - s.b;        // Undefined, different array objects
(s.a + 3) - s.a; // OK, same array object
```

Definition 6.3.2. *The judgment* $\Gamma, m \vdash (a_1 \text{ - } a_2)\ \mathsf{defined}$ *describes if* subtraction of an address a_2 from a_1 *is defined. It is inductively defined as:*

$$\frac{\overline{m} \vdash o\ \mathsf{alive} \qquad |\,\vec{r}_1\,|_\circ = |\,\vec{r}_2\,|_\circ \qquad ((i_1 - i_2) \div \mathsf{sizeof}_\Gamma\ \sigma_\mathsf{p}) : \mathsf{signed\ ptr\_rank}}{\Gamma, m \vdash ((o : \tau, \vec{r}_1, i_1)_{\sigma >_* \sigma_\mathsf{p}} \text{ - } (o : \tau, \vec{r}_2, i_2)_{\sigma >_* \sigma_\mathsf{p}})\ \mathsf{defined}}$$

The result of the subtraction $a_1 \mathbin{\mathtt{-}} a_2$ *is defined as follows:*

$$(o : \tau, \vec{r}_1, i_1)_{\sigma >_* \sigma_{\mathsf{p}}} \mathbin{\mathtt{-}} (o : \tau, \vec{r}_2, i_2)_{\sigma >_* \sigma_{\mathsf{p}}} := (i_1 - i_2) \div \mathsf{sizeof}_\Gamma\ \sigma_{\mathsf{p}}$$

The side-condition $((i_1 - i_2) \div \mathsf{sizeof}_\Gamma\ \sigma_{\mathsf{p}}) : \mathsf{signed\ ptr\_rank}$ ensures that the difference can be represented as an integer of type `ptrdiff_t` [ISO12, 6.5.6p9].

6.3.3 Pointer addition

Pointer addition can be used to move a pointer through an array object. Overflow of pointer addition has undefined behavior even if the overflown pointer is not dereferenced [ISO12, 6.5.6p8]. For example:

```
int a[10], *p = a + 15;  // Undefined, overflow
```

Definition 6.3.3. *The judgment* $\Gamma, m \vdash (a \mathbin{\mathtt{+}} j)\ \mathsf{defined}$ *describes if* addition of an integer $j \in \mathbb{Z}$ to an address a is defined*. It is inductively defined as:*

$$\frac{\overline{m} \vdash o\ \mathsf{alive} \qquad 0 \leq i + j \cdot \mathsf{sizeof}_\Gamma\ \sigma_{\mathsf{p}} \leq \mathsf{sizeof}_\Gamma\ \sigma \cdot \mathsf{size}\ \vec{r}}{\Gamma, m \vdash ((o : \tau, \vec{r}, i)_{\sigma >_* \sigma_{\mathsf{p}}} \mathbin{\mathtt{+}} j)\ \mathsf{defined}}$$

The result of the addition $a \mathbin{\mathtt{+}} j$ *is defined as follows:*

$$(o : \tau, \vec{r}, i)_{\sigma >_* \sigma_{\mathsf{p}}} \mathbin{\mathtt{+}} j := (o : \tau, \vec{r}, i + j \cdot \mathsf{sizeof}_\Gamma\ \sigma_{\mathsf{p}})_{\sigma >_* \sigma_{\mathsf{p}}}$$

Pointer addition is *type aware*, which means that adding j to an address of type σ increases its byte offset by $j \cdot \mathsf{sizeof}_\Gamma\ \sigma$ instead of j.

6.3.4 Pointer casting

Casting an pointer has undefined behavior in case the resulting pointer is ill-aligned or if the cast breaks dynamic typing. For example:

```
int x;
(short*)(void*)&x;                // Undefined, int* cast to short*
(int*)((unsigned char*)&x + 1); // Undefined, ill-aligned
```

Definition 6.3.4. *The judgment* $\Gamma, m \vdash (\sigma_{\mathsf{p}})a\ \mathsf{defined}$ *describes if* casting an address a to pointer type σ_{p} is defined*. It is inductively defined as:*

$$\frac{\overline{m} \vdash o\ \mathsf{alive} \qquad \mathsf{sizeof}_\Gamma\ \sigma_{\mathsf{p}} \mid i \qquad \sigma >_* \sigma_{\mathsf{p}}}{\Gamma, m \vdash (\sigma_{\mathsf{p}})(o : \tau, \vec{r}, i)_{\sigma >_* \sigma'_{\mathsf{p}}}\ \mathsf{defined}}$$

The result of the cast $(\sigma_{\mathsf{p}})a$ *is defined as follows:*

$$(\sigma_{\mathsf{p}})(o : \tau, \vec{r}, i)_{\sigma >_* \sigma'_{\mathsf{p}}} := (o : \tau, \vec{r}, i)_{\sigma >_* \sigma_{\mathsf{p}}}$$

6.3.5 Value operators

This section lifts the previously defined semantics of unary, binary and cast operators on addresses (Sections 6.3.1-6.3.4) and integers (Section 3.2) to abstract values.

Definition 6.3.5. *The judgment* $m \vdash (\circledcirc_u\, v)\ \mathsf{defined}$ *describes if applying a* unary operator $\circledcirc_u$ to a value v is defined. *The result is denoted by* $\circledcirc_u\, v$. *Both notions are defined by lifting the notions on integers and addresses to abstract values:*

$\circledcirc_u$	v	$m \vdash (\circledcirc_u\, v)\ \mathsf{defined}$	$\circledcirc_u\, v$
$-$	$\mathsf{int}_{\tau_\mathsf{i}}\, x$	$\mathsf{int\_arithop\_ok}\,(-)\ 0\ \tau_\mathsf{i}\ x\ \tau_\mathsf{i}$	$\mathsf{int}_{\lceil \tau_\mathsf{i} \rceil}\ (\mathsf{int\_arithop}\,(-)\ 0\ \tau_\mathsf{i}\ x\ \tau_\mathsf{i})$
$\sim$	$\mathsf{int}_{\tau_\mathsf{i}}\, x$	True	$\mathsf{int}_{\lceil \tau_\mathsf{i} \rceil}\ ((\sim \overline{x : \lceil \tau_\mathsf{i} \rceil})_{\lceil \tau_\mathsf{i} \rceil})$
$!$	$\mathsf{int}_{\tau_\mathsf{i}}\, x$	True	$\mathsf{int}_{\mathsf{signed\ int}}\ (\text{if } x = 0 \text{ then } 1 \text{ else } 0)$
$!$	$\mathsf{ptr}\, a$	$\overline{m} \vdash a\ \mathsf{alive}$	$\mathsf{int}_{\mathsf{signed\ int}}\ 0$
$!$	$\mathsf{ptr}\,(\mathsf{NULL}\ \tau_\mathsf{p})$	True	$\mathsf{int}_{\mathsf{signed\ int}}\ 1$
$!$	$\mathsf{ptr}\,(f^{\vec{\tau} \mapsto \tau})$	True	$\mathsf{int}_{\mathsf{signed\ int}}\ 0$

The integer encoding functions $\overline{\_ : \tau_\mathsf{i}} : \mathbb{Z} \to \mathsf{list\ bool}$ *and* $(\_)_{\tau_\mathsf{i}} : \mathsf{list\ bool} \to \mathbb{Z}$ *are defined in Definition 3.1.4 on page 41. The predicate* int_arithop_ok *and function* int_arithop *are part of the integer environment as defined in Definition 3.2.5 on page 43.*

The judgments $\Gamma, m \vdash (v_1 \circledcirc v_2)\ \mathsf{defined}$ and $\Gamma, m \vdash (\tau)v\ \mathsf{defined}$ for binary operators and casts, and their resulting values $v_1 \circledcirc v_2$ and $(\tau)v$, are defined similarly. We refer the interested reader to the Coq formalization for the details.

6.3.6 Conditionals

In order to define which branch of a conditional expression $e\ ?\ e_2 : e_3$ or conditional statement $\mathsf{if}\ (e)\ s_1\ \mathsf{else}\ s_2$ has to be taken, we define what is means for a value "to be zero". Due to typing, we know that the controlling expression e has integer or pointer type, so we only need to define what it means for values of those types to be zero. Branching on an indeterminate value has undefined behavior.

Definition 6.3.6. *The judgment* $\mathsf{zero}\ v_\mathsf{b}$ *denotes that a base value* v_b is zero. *It only has a defined meaning if* $m \vdash (\mathsf{zero}\ v_\mathsf{b})\ \mathsf{defined}$ *holds. These judgments are inductively defined as:*

$$\frac{}{m \vdash (\mathsf{zero}\ (\mathsf{int}_{\tau_\mathsf{i}}\, x))\ \mathsf{defined}} \qquad \frac{m \vdash p\ \mathsf{alive}}{m \vdash (\mathsf{zero}\ (\mathsf{ptr}\, p))\ \mathsf{defined}}$$

$$\frac{}{\mathsf{zero}\ (\mathsf{int}_{\tau_\mathsf{i}}\, 0)} \qquad \frac{}{\mathsf{zero}\ (\mathsf{ptr}\,(\mathsf{NULL}\ \sigma_\mathsf{p}))}$$

6.3.7 Assignments

This section describes the semantics of the various kinds of assignments.

Definition 6.3.7. *The judgment* $\Gamma, m \vdash (a\ \alpha\ v) \mapsto_{\mathsf{ass}} (v_a,\ v')$ *describes the* assigned value v_a and resulting r-value v' of an assignment $a\ \alpha\ v$. *It is inductively defined as:*

$$\frac{\Gamma, m \vdash (\tau)v\ \mathsf{defined} \qquad \tau = \mathsf{typeof}\ a}{\Gamma, m \vdash (a := v) \mapsto_{\mathsf{ass}} ((\tau)v,\ (\tau)v)}$$

$$\frac{m\langle a\rangle_\Gamma = v_a \qquad \Gamma, m \vdash (v_a \circledcirc v)\ \mathsf{defined} \qquad \Gamma, m \vdash (\tau)(v_a \circledcirc v)\ \mathsf{defined} \qquad \tau = \mathsf{typeof}\ a}{\Gamma, m \vdash (a \circledcirc:= v) \mapsto_{\mathsf{ass}} ((\tau)(v_a \circledcirc v),\ (\tau)(v_a \circledcirc v))}$$

$$\frac{m\langle a\rangle_\Gamma = v_a \qquad \Gamma, m \vdash (v_a \circledcirc v)\ \mathsf{defined} \qquad \Gamma, m \vdash (\tau)(v_a \circledcirc v)\ \mathsf{defined} \qquad \tau = \mathsf{typeof}\ a}{\Gamma, m \vdash (a :=\circledcirc v) \mapsto_{\mathsf{ass}} ((\tau)(v_a \circledcirc v),\ v_a)}$$

Note that the casts are needed to take the integer promotions [ISO12, 6.3.1.1p2] and the usual arithmetic conversions [ISO12, 6.3.1.8p1] into account.

6.3.8 Address indexing

This section defines the *top address* $\mathsf{top}_\tau\ o$, which refers to the corresponding object with object identifier o in memory, and the *element operation* $a\langle r\rangle_\Gamma$, which selects an element r of an address a that refers to an object of array, struct or union type. Let us demonstrate where these operations are used:

```
struct S {
  union U { signed char x[2]; int y; } u; void *p;
} s;
```

In the context of this type declaration, we consider the pointer `s.u.x + 2`, which corresponds to the following expression in CH$_2$O core C:

$$\overbrace{\left(\&\left(\underbrace{x_{\mathtt{s}}\ ._{\imath}\ (\xhookrightarrow{\text{struct S}} 0)\ ._{\imath}\ (\xhookrightarrow{\text{union U}}_{\bullet}\ 0)}_{\text{l-value, }\mathtt{s.u.x}}\ ._{\imath}\ (\xhookrightarrow{\text{signed int[2]}} 0)\right) + 2\right)}^{\text{r-value, }\mathtt{s.u.x\ +\ 2}}$$

The operators $._{\imath}\ (\xhookrightarrow{\text{signed int[2]}} 0)$ and & make the conversion of the l-value `s.u.x` of array type into a pointer to its first element explicit.

In the operational semantics, we interpret the variable $x_{\mathtt{s}}$ as the address $\mathsf{top}_o\,\tau$ where the object identifier o and type τ are obtained from the stack (Definition 6.1.10). The expression $(e\ ._{\imath}\ r)$ is interpreted using the operation $a\langle r\rangle_\Gamma$.

Definition 6.3.8. *The* top address $\mathsf{top} : \mathsf{index} \to \mathsf{type} \to \mathsf{addr}$ *is defined as:*

$$\mathsf{top}_\tau\ o := (o : \tau, \varepsilon, 0)_{\tau >_* \tau}.$$

Definition 6.3.9. *The* element operation $(\_)\langle\_\rangle_\Gamma : \mathsf{addr} \to \mathsf{refseg} \to \mathsf{addr}$ *on addresses is defined as:*

$$((o : \tau, \vec{r}, i)_{\sigma >_* \_})\langle r\rangle_\Gamma := (o : \tau, \vec{r}_2, \mathsf{sizeof}_\Gamma\ \sigma_2 \cdot \mathsf{offset}\ r)_{\sigma_2 >_* \sigma_2}$$

where $\vec{r}_2 := (\mathsf{setoffset}\,(i \div \mathsf{sizeof}_\Gamma\ \sigma)\ \vec{r})\,(\mathsf{setoffset}\,0\ r)$. *The element operation is only well-defined in case a type* σ_2 *with* $\Gamma \vdash r : \sigma \rightarrowtail \sigma_2$ *exists.*

The element operation involves some fiddling with byte offsets due to our treatment of end-of-array and **`unsigned char*`** pointers. It enjoys nice properties on strict addresses (addresses that are not end-of-array).

Lemma 6.3.10. *If* $\Gamma, \Delta \vdash a : \tau$, $\Gamma \vdash r : \tau \rightarrowtail \sigma$, *and* a *is a strict address, then:*

$$\mathsf{index}\ (a\langle r\rangle_\Gamma) = \mathsf{index}\ a \qquad \mathsf{ref}_\Gamma\ (a\langle r\rangle_\Gamma) = (\mathsf{ref}_\Gamma\ a) r \qquad \mathsf{byte}_\Gamma\ (a\langle r\rangle_\Gamma) = 0.$$

The functions $\mathsf{ref}_\Gamma : \mathsf{addr} \to \mathsf{ref}$ *and* $\mathsf{byte}_\Gamma : \mathsf{addr} \to \mathbb{N}$ *yield the normalized reference and normalized byte offset of an address (Definition 5.2.10 on page 68).*

6.3.9 Value indexing

This section defines the operation $v[\vec{r}]_\Gamma$, which yields the subvalue at location $\vec{r}$ in the abstract value v. It is used to describe the semantics of the expression construct $e\ ._{\mathsf{1}}\ r$ for indexing of r-values, which is used to select a field of a struct or union returned by a function. It corresponds to `f().x` in the following example:

```
struct S { int x; int y; };
struct S f() { struct S r = { 2, 3 }; return r; }
int main() { return f().x; }
```

Definition 6.3.11. *The* lookup operation on abstract values $(\_)[\_]_\Gamma : \mathsf{val} \to \mathsf{ref} \to \mathsf{option}\ \mathsf{val}$ *is defined as:*

$$\begin{aligned}
v[\varepsilon]_\Gamma &:= v \\
(\mathsf{array}_\tau\ \vec{v})[(\xhookrightarrow{\tau[n]} i)\ \vec{r}]_\Gamma &:= v_i[\vec{r}]_\Gamma \\
(\mathsf{struct}_t\ \vec{v})[(\xhookrightarrow{\mathsf{struct}\ t} i)\ \vec{r}]_\Gamma &:= v_i[\vec{r}]_\Gamma \\
(\mathsf{union}_t\ (i, v))[(\xhookrightarrow{\mathsf{union}\ t}_q i)\ \vec{r}]_\Gamma &:= v[\vec{r}]_\Gamma \\
(\mathsf{union}_t\ (j, v))[(\xhookrightarrow{\mathsf{union}\ t}_q i)\ \vec{r}]_\Gamma &:= \begin{cases} v[\vec{r}]_\Gamma & \text{if } i = j \\ ((\overline{v}^{\Gamma} \lightning^{\infty})_{[0,\, s)})^{\tau_i}_{\Gamma}\,[\vec{r}]_\Gamma & \text{if } i \neq j \text{ and } q = \bullet \\ \bot & \text{if } i \neq j \text{ and } q = \circ \end{cases} \\
&\ \text{where } \Gamma\, t = \vec{\tau} \text{ and } s := \mathsf{bitsizeof}_\Gamma\, \tau_i \\
(\overline{\mathsf{union}}_t\ \vec{v})[(\xhookrightarrow{\mathsf{union}\ t}_q i)\ \vec{r}]_\Gamma &:= v_i[\vec{r}]_\Gamma
\end{aligned}$$

The definition of $v[\vec{r}]_\Gamma$ is similar to the definition of the lookup operation $w[\vec{r}]_\Gamma$ on memory trees (Definition 5.4.10 on page 76). For all cases, except for unions, it performs a straightforward subtree lookup. If a union is assessed through a pointer to a different variant than its current variant, the value is reinterpreted as a value of another type by conversion into bits and back.

6.3.10 Compound literals

Recall from page 6.1 in Section 6.1 that initializers and compound literals are encoded as sequences:

$$\mathbf{0}^{\tau}_{\Gamma}[\vec{r}_0 := e_0] \dots [\vec{r}_{n-1} := e_{n-1}]$$

where $\mathbf{0}^{\tau}_{\Gamma}$ denotes the value of type τ that is initialized with zeros.

This section defines the zero value $\mathbf{0}^{\tau}_{\Gamma}$ and the insert operation $v_1[\vec{r} := v_2]_{\Gamma}$ on abstract values corresponding to the $e_1[\vec{r} := e_2]$ operator. The definition of $\mathbf{0}^{\tau}_{\Gamma}$ corresponds to [ISO12, 6.7.9] of the C11 standard.

Definition 6.3.12. *The operation* $\mathbf{0}^{(\_)}_{\Gamma} : \mathsf{type} \to \mathsf{val}$ *constructs* the value initialized with zeros. *It is defined as:*

$$\begin{aligned}
\mathbf{0}^{\mathsf{void}}_{\Gamma} &:= \mathsf{nothing} \qquad \mathbf{0}^{\tau_{\mathsf{i}}}_{\Gamma} := \mathsf{int}_{\tau_{\mathsf{i}}}\, 0 \qquad \mathbf{0}^{\tau_{\mathsf{p}}*}_{\Gamma} := \mathsf{ptr}\,(\mathsf{NULL}\ \tau_{\mathsf{p}})\\
\mathbf{0}^{\tau[n]}_{\Gamma} &:= \mathsf{array}_{\tau}\,(\mathbf{0}^{\tau}_{\Gamma})^{n}\\
\mathbf{0}^{\mathsf{struct}\ t}_{\Gamma} &:= \mathsf{struct}_{t}\,(\mathbf{0}^{\tau_0}_{\Gamma} \dots \mathbf{0}^{\tau_{n-1}}_{\Gamma}) && \text{if } \Gamma\, t = \vec{\tau} \text{ and } n = |\vec{\tau}|\\
\mathbf{0}^{\mathsf{union}\ t}_{\Gamma} &:= \mathsf{union}_{t}\,(0, \mathbf{0}^{\tau_0}_{\Gamma}) && \text{if } \Gamma\, t = \vec{\tau}
\end{aligned}$$

The semantics of $e_1[\vec{r} := e_2]$ is defined in terms of the insert operation $v_1[\vec{r} := v_2]_{\Gamma}$ on abstract values, which replaces the subvalue at location $\vec{r}$ in v_1 by v_2.

Definition 6.3.13. *The* insert operation on abstract values $(\_)[\_ := \_]_{\Gamma} : \mathsf{ref} \to \mathsf{val} \to \mathsf{option\ val}$ *is defined as:*

$$\begin{aligned}
v[\varepsilon := v']_{\Gamma} &:= v'\\
(\mathsf{array}_{\tau}\,\vec{v})[(\xhookrightarrow{\tau[n]} i)\,\vec{r} := v']_{\Gamma} &:= \mathsf{array}_{\tau}\,(\vec{v}[i := v_i[\vec{r} := v']_{\Gamma}])\\
(\mathsf{struct}_{t}\,\vec{v})[(\xhookrightarrow{\mathsf{struct}\ i} t)\,\vec{r} := v']_{\Gamma} &:= \mathsf{struct}_{t}\,(\vec{v}[i := v_i[\vec{r} := v']_{\Gamma}])\\
(\mathsf{union}_{t}\,(i, v))[(\xhookrightarrow{\mathsf{union}\ t}_{q} j)\,\vec{r} := v']_{\Gamma} &:= \begin{cases} \mathsf{union}_{t}\,(i, v[\vec{r} := v']_{\Gamma}) & \text{if } i = j\\ \mathsf{union}_{t}\,(i, ((\overline{v}^{\Gamma}\,\lightning^{\infty})_{[0,\,s)})^{\tau_i}_{\Gamma}\,[\vec{r} := v']_{\Gamma}) & \text{if } i \neq j \end{cases}\\
&\ \text{where } \Gamma\, t = \vec{\tau} \text{ and } s := \mathsf{bitsizeof}_{\Gamma}\,\tau_i\\
(\overline{\mathsf{union}}_{t}\,\vec{v})[(\xhookrightarrow{\mathsf{union}\ t}_{q} i)\,\vec{r} := v']_{\Gamma} &:= \mathsf{union}_{t}\,(i, v_i[\vec{r} := v']_{\Gamma})
\end{aligned}$$

The result of $v[\vec{r} := v']_{\Gamma}$ *is only well-defined in case* $v[\vec{r}]_{\Gamma} \neq \bot$.

6.4 Operational semantics of expressions

This section defines the head reduction $(e_1, m_1) \rightarrow_{\mathsf{h}} (e_2, m_2)$ of expressions, which in turn will be lifted to full expressions using *evaluation contexts.*

Definition 6.4.1. Head reduction $\Gamma, \rho \vdash (e_1, m_1) \rightarrow_{\mathsf{h}} (e_2, m_2)$ of expressions *is inductively defined by the following rules:*

1. $(x_i, m) \rightarrow_{\mathsf{h}} ([\mathsf{top}_{\tau}\ o]_{\emptyset}, m)$ *if* $\rho(i) = (o,\ \tau)$

2. $(*[\mathsf{ptr}\,p]_\Omega, m) \mathrel{-\!\!\rhd_{\mathsf{h}}} ([p]_\Omega, m)$ *if* $\overline{m} \vdash p$ alive
3. $(\&[p]_\Omega, m) \mathrel{-\!\!\rhd_{\mathsf{h}}} ([\mathsf{ptr}\,p]_\Omega, m)$
4. $([a]_\Omega \;\boldsymbol{.}_{\mathbf{l}}\; r, m) \mathrel{-\!\!\rhd_{\mathsf{h}}} ([a\langle r\rangle_\Gamma]_\Omega, m)$ *if* $\Gamma \vdash a$ strict
5. $([v]_\Omega \;\boldsymbol{.}_{\mathbf{r}}\; r, m) \mathrel{-\!\!\rhd_{\mathsf{h}}} ([v']_\Omega, m)$ *if* $v[r]_\Gamma = v'$
6. $([v_1]_{\Omega_1}[\vec{r} := [v_2]_{\Omega_2}], m) \mathrel{-\!\!\rhd_{\mathsf{h}}} ([v_1[\vec{r} := v_2]_\Gamma]_{\Omega_1 \cup \Omega_2}, m)$ *if* $v_1[\vec{r}]_\Gamma \neq \bot$
7. $([a]_{\Omega_1}\; \alpha\; [v]_{\Omega_2}, m) \mathrel{-\!\!\rhd_{\mathsf{h}}} ([v']_{\{a\}_\Gamma \cup \Omega_1 \cup \Omega_2}, \mathsf{lock}_\Gamma\, a\, (m\langle a := v_a\rangle_\Gamma))$
   if $\mathsf{writable}_\Gamma\, a\, m$ *and* $\Gamma, m \vdash (a\; \alpha\; v) \mapsto_{\mathsf{ass}} (v_a,\, v')$
8. $(\mathsf{load}\; [a]_\Omega, m) \mathrel{-\!\!\rhd_{\mathsf{h}}} ([v]_\Omega, \mathsf{force}_\Gamma\, a\, m)$ *if* $m\langle a\rangle_\Gamma = v$
9. $(\mathsf{alloc}_\tau\; [\mathsf{int}_{\tau_{\mathsf{i}}}\, n]_\Omega, m) \mathrel{-\!\!\rhd_{\mathsf{h}}} ([\mathsf{NULL}\;\tau]_\Omega, m)$ *if* $0 < n$.
10. $(\mathsf{alloc}_\tau\; [\mathsf{int}_{\tau_{\mathsf{i}}}\, n]_\Omega, m) \mathrel{-\!\!\rhd_{\mathsf{h}}} ([(\mathsf{top}_{\tau[n]}\, o)\langle \xhookrightarrow{\tau[n]} 0\rangle_\Gamma]_\Omega, \mathsf{alloc}_\Gamma\, o\, (\tau[n])\; \mathsf{true}\; m)$
    if $0 < n$, *for any* $o \notin \mathsf{dom}\; m$
11. $(\mathsf{free}\; [a]_\Omega, m) \mathrel{-\!\!\rhd_{\mathsf{h}}} ([\mathsf{nothing}]_\Omega, \mathsf{free}\; (\mathsf{index}\; a)\; m)$ *if* $\mathsf{freeable}\; a\; m$
12. $(\circledcirc_u\; [v]_\Omega, m) \mathrel{-\!\!\rhd_{\mathsf{h}}} ([\circledcirc_u\, v]_\Omega, m)$ *if* $m \vdash (\circledcirc_u\, v)$ defined
13. $([v_1]_{\Omega_1} \circledcirc [v_2]_{\Omega_2}, m) \mathrel{-\!\!\rhd_{\mathsf{h}}} ([v_1 \circledcirc v_2]_{\Omega_1 \cup \Omega_2}, m)$ *if* $\Gamma, m \vdash (v_1 \circledcirc v_2)$ defined
14. $((\tau)[v]_\Omega, m) \mathrel{-\!\!\rhd_{\mathsf{h}}} ([(\tau)v]_\Omega, m)$ *if* $\Gamma, m \vdash (\tau)v$ defined
15. $([v_{\mathsf{b}}]_\Omega \;?\; e_2 : e_3, m) \mathrel{-\!\!\rhd_{\mathsf{h}}} (e_2, \mathsf{unlock}\;\Omega\; m)$ *if* $m \vdash (\mathsf{zero}\; v_{\mathsf{b}})$ defined *and* $\neg\mathsf{zero}\; v_{\mathsf{b}}$
16. $([v_{\mathsf{b}}]_\Omega \;?\; e_2 : e_3, m) \mathrel{-\!\!\rhd_{\mathsf{h}}} (e_3, \mathsf{unlock}\;\Omega\; m)$ *if* $m \vdash (\mathsf{zero}\; v_{\mathsf{b}})$ defined *and* $\mathsf{zero}\; v_{\mathsf{b}}$
17. $(([v]_\Omega, e), m) \mathrel{-\!\!\rhd_{\mathsf{h}}} (e, \mathsf{unlock}\;\Omega\; m)$

Notice that the alloc function non-deterministically returns a `NULL` pointer (rule 9) or a pointer to a newly allocated object (rule 10). A `NULL` pointer is returned in case the system has run out of memory, see also Section 2.6.3.

We use evaluation contexts [FFKD87] to lift the head reduction to full expressions. Evaluation contexts provide a way to compactly describe the selection of head redexes as part of a whole expression.

Definition 6.4.2. (Singular) expression contexts *are inductively defined as:*

$$\begin{aligned}
\mathcal{E}_{\mathsf{s}} \in \mathsf{ectx}_{\mathsf{s}} ::=\ & *\square \mid \&\square \mid \square \;\boldsymbol{.}_{\mathbf{l}}\; r \mid \square \;\boldsymbol{.}_{\mathbf{r}}\; r \mid \square[\vec{r} := e_2] \mid e_1[\vec{r} := \square] \\
& \mid \square\; \alpha\; e_2 \mid e_1\; \alpha\; \square \mid \mathsf{load}\; \square \mid \square(\vec{e}) \mid e(\vec{e}_1\, \square\, \vec{e}_2) \mid \mathsf{alloc}_\tau\; \square \mid \mathsf{free}\; \square \\
& \mid\; \circledcirc_u\; \square \mid \square \circledcirc e_2 \mid e_1 \circledcirc \square \mid (\tau)\square \mid \square \;?\; e_2 : e_3 \mid (\square, e_2) \\
\mathcal{E} \in \mathsf{ectx} :=\ & \mathsf{list}\; \mathsf{ectx}_{\mathsf{s}}
\end{aligned}$$

Given an expression context $\mathcal{E}$ *and an expression* e, *the* substitution of e for the hole $\square$ in $\mathcal{E}$, *notation* $\mathcal{E}[\,e\,]$, *is defined as usual.*

The typing judgment $\Gamma, \Delta, \vec{\tau} \vdash \mathcal{E} : \tau_{\mathsf{lr}} \rightarrowtail \sigma_{\mathsf{lr}}$ for expression contexts is derived from the typing judgment $\Gamma, \Delta, \vec{\tau} \vdash e : \tau_{\mathsf{lr}}$ for expressions (Definition 6.1.7) in the obvious way. We define it explicitly for completeness' sake.

Definition 6.4.3. *The judgment* $\Gamma, \Delta, \vec{\tau} \vdash \mathcal{E}_{\mathsf{s}} : \tau_{\mathsf{lr}} \rightarrowtail \sigma_{\mathsf{lr}}$ *describes that* $\mathcal{E}_{\mathsf{s}}$ *is a* valid singular expression context of type σ_{lr} with a hole $\Box$ of type τ_{lr}. *It is inductively defined as:*

$$\frac{}{\Gamma, \Delta, \vec{\tau} \vdash *\Box : (\tau*)_{\mathbf{r}} \rightarrowtail (\tau)_{\mathbf{l}}} \qquad \frac{}{\Gamma, \Delta, \vec{\tau} \vdash \&\Box : (\tau_{\mathsf{p}})_{\mathbf{l}} \rightarrowtail (\tau_{\mathsf{p}}*)_{\mathbf{r}}}$$

$$\frac{\Gamma \vdash r : \tau \rightarrowtail \sigma}{\Gamma, \Delta, \vec{\tau} \vdash \Box \;._{\mathbf{l}}\; r : \tau_{\mathbf{l}} \rightarrowtail \sigma_{\mathbf{l}}} \qquad \frac{\Gamma \vdash r : \tau \rightarrowtail \sigma}{\Gamma, \Delta, \vec{\tau} \vdash \Box \;._{\mathbf{r}}\; r : \tau_{\mathbf{r}} \rightarrowtail \sigma_{\mathbf{r}}}$$

$$\frac{\Gamma \vdash \vec{r} : \tau \rightarrowtail \sigma \qquad \Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e_2 : \sigma}{\Gamma, \Delta, \vec{\tau} \vdash \Box[\vec{r} := e_2] : \tau_{\mathbf{r}} \rightarrowtail \tau_{\mathbf{r}}} \qquad \frac{\Gamma \vdash \vec{r} : \tau \rightarrowtail \sigma \qquad \Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e_1 : \tau}{\Gamma, \Delta, \vec{\tau} \vdash e_1[\vec{r} := \Box] : \sigma_{\mathbf{r}} \rightarrowtail \tau_{\mathbf{r}}}$$

$$\frac{\tau_1 \;\alpha\; \tau_2 : \sigma \qquad \Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e_2 : \tau_2}{\Gamma, \Delta, \vec{\tau} \vdash \Box \;\alpha\; e_2 : (\tau_1)_{\mathbf{l}} \rightarrowtail \sigma_{\mathbf{r}}} \qquad \frac{\tau_1 \;\alpha\; \tau_2 : \sigma \qquad \Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{l}} e_1 : \tau_1}{\Gamma, \Delta, \vec{\tau} \vdash e_1 \;\alpha\; \Box : (\tau_2)_{\mathbf{r}} \rightarrowtail \sigma_{\mathbf{r}}}$$

$$\frac{\mathsf{complete}_{\Gamma}\; \tau}{\Gamma, \Delta, \vec{\tau} \vdash \mathsf{load}\; \Box : \tau_{\mathbf{l}} \rightarrowtail \tau_{\mathbf{r}}} \qquad \frac{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} \vec{e} : \vec{\sigma} \qquad \mathsf{complete}_{\Gamma}\; \sigma}{\Gamma, \Delta, \vec{\tau} \vdash \Box(\vec{e}) : ((\vec{\sigma} \to \sigma)*)_{\mathbf{r}} \rightarrowtail \sigma_{\mathbf{r}}}$$

$$\frac{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e : (\vec{\sigma}_1 \sigma \vec{\sigma}_2 \to \tau)* \quad \Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} \vec{e}_1 : \vec{\sigma}_1 \quad \Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} \vec{e}_2 : \vec{\sigma}_2 \quad \mathsf{complete}_{\Gamma}\; \tau}{\Gamma, \Delta, \vec{\tau} \vdash e(\vec{e}_1 \,\Box\, \vec{e}_2) : \sigma_{\mathbf{r}} \rightarrowtail \tau_{\mathbf{r}}}$$

$$\frac{\Gamma \vdash \tau}{\Gamma, \Delta, \vec{\tau} \vdash \mathsf{alloc}_{\tau}\; \Box : (\tau_{\mathsf{i}})_{\mathbf{r}} \rightarrowtail \tau_{\mathbf{l}}} \qquad \frac{}{\Gamma, \Delta, \vec{\tau} \vdash \mathsf{free}\; \Box : (\tau_{\mathsf{p}})_{\mathbf{l}} \rightarrowtail \mathsf{void}_{\mathbf{r}}}$$

$$\frac{\circledcirc_u \tau : \sigma}{\Gamma, \Delta, \vec{\tau} \vdash \circledcirc_u \Box : \tau_{\mathbf{r}} \rightarrowtail \sigma_{\mathbf{r}}} \qquad \frac{\tau_1 \circledcirc \tau_2 : \sigma \qquad \Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e_2 : \tau_2}{\Gamma, \Delta, \vec{\tau} \vdash \Box \circledcirc e_2 : (\tau_1)_{\mathbf{r}} \rightarrowtail \sigma_{\mathbf{r}}}$$

$$\frac{\tau_1 \circledcirc \tau_2 : \sigma \qquad \Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e_1 : \tau_1}{\Gamma, \Delta, \vec{\tau} \vdash e_1 \circledcirc \Box : (\tau_2)_{\mathbf{r}} \rightarrowtail \sigma_{\mathbf{r}}} \qquad \frac{(\sigma)\tau : \sigma \qquad \Gamma \vdash \sigma}{\Gamma, \Delta, \vec{\tau} \vdash (\sigma)\Box : \tau_{\mathbf{r}} \rightarrowtail \sigma_{\mathbf{r}}}$$

$$\frac{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e_2 : \tau_2}{\Gamma, \Delta, \vec{\tau} \vdash (\Box, e_2) : (\tau_1)_{\mathbf{r}} \rightarrowtail (\tau_2)_{\mathbf{r}}} \qquad \frac{\tau_{\mathsf{b}} \neq \mathsf{void} \qquad \Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e_2 : \sigma \qquad \Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e_3 : \sigma}{\Gamma, \Delta, \vec{\tau} \vdash \Box \;?\; e_2 : e_3 : (\tau_{\mathsf{b}})_{\mathbf{r}} \rightarrowtail \sigma_{\mathbf{r}}}$$

Definition 6.4.4. *The judgment* $\Gamma, \Delta, \vec{\tau} \vdash \mathcal{E} : \tau_{\mathsf{lr}} \rightarrowtail \sigma_{\mathsf{lr}}$ *describes that* $\mathcal{E}$ *is a* valid expression context of type σ_{lr} with a hole $\Box$ of type τ_{lr}. *It is inductively defined as:*

$$\frac{}{\Gamma, \Delta, \vec{\tau} \vdash \varepsilon : \sigma_{\mathsf{lr}} \rightarrowtail \sigma_{\mathsf{lr}}} \qquad \frac{\Gamma, \Delta, \vec{\tau} \vdash \mathcal{E}_{\mathsf{s}} : \tau_{\mathsf{lr}} \rightarrowtail \sigma_{\mathsf{lr}} \qquad \Gamma, \Delta, \vec{\tau} \vdash \mathcal{E} : \sigma_{\mathsf{lr}} \rightarrowtail \upsilon_{\mathsf{lr}}}{\Gamma, \Delta, \vec{\tau} \vdash \mathcal{E}_{\mathsf{s}}\, \mathcal{E} : \tau_{\mathsf{lr}} \rightarrowtail \upsilon_{\mathsf{lr}}}$$

Lemma 6.4.5. *Validity of expression contexts enjoys the substitution property:*

$$\Gamma, \Delta, \vec{\tau} \vdash \mathcal{E}[\,e\,] : \sigma_{\mathsf{lr}} \quad \text{iff} \quad \Gamma, \Delta, \vec{\tau} \vdash \mathcal{E} : \tau_{\mathsf{lr}} \rightarrowtail \sigma_{\mathsf{lr}} \text{ and } \Gamma, \Delta, \vec{\tau} \vdash e : \tau_{\mathsf{lr}} \text{ for some } \tau_{\mathsf{lr}}.$$

Intuitively, an expression context $\mathcal{E}$ is just an expression with a hole $\Box$ that describes the location of the head redex. The head reduction is lifted to full expressions as follows (see Rule 2a of Definition 6.5.8 for the actual rule):

$$\frac{(e_1, m_1) \rightarrowtriangle_{\mathsf{h}} (e_2, m_2)}{(\mathcal{E}[\,e_1\,], m_1) \rightarrowtriangle (\mathcal{E}[\,e_2\,], m_2)}$$

We follow Norrish [Nor98], Leroy [Ler09b] and Ellison and Roşu [ER12b] by using expression contexts to formalize the unspecified order of expression execution. Since expression contexts overlap, non-trivial expressions e can be decomposed into multiple combinations of head redexes e' and expression contexts $\mathcal{E}$ that enjoy $e = \mathcal{E}[\,e'\,]$. For example, the expression $e_1 + e_2$ can be written as both $(\square + e_2)[\,e_1\,]$ and $(e_1 + \square)[\,e_2\,]$. The above reduction rule can therefore be applied in two different ways, allowing one to reduce the subexpressions e_1 and e_2 in any order.

The execution order of expressions is however not unspecified in all aspects by the C11 standard. The first operand of the conditional $e_1 \mathrel{?} e_2 : e_3$ and comma (e_1, e_2) expression is guaranteed to be evaluated to a value before (one of) the other operands are evaluated at all[1]. We therefore do *not* include the expression contexts $e_1 \mathrel{?} \square : e_3$, $e_1 \mathrel{?} e_2 : \square$ and $(e_1, \square)$.

Undefined behavior and function calls make the actual reduction rules for full expressions somewhat more complicated. We follow Leroy [Ler09b] by distinguishing the following three cases (see Definition 6.5.8 for the formal definition as part of the whole program semantics):

- **Subexpressions that can head reduce**. This is just the rule that we have described above: a whole expression $\mathcal{E}[\,e_1\,]$ may reduce to $\mathcal{E}[\,e_2\,]$ provided that we have $(e_1, m_1) \rightharpoonup_{\mathsf{h}} (e_2, m_2)$.
- **Subexpressions that contain a function call.** Subexpressions of the shape $[\nu]_\Omega([\vec{\nu}]_{\vec{\Omega}})$, *i.e.* function calls, are not described by the head reduction. Instead, a function call suspends expression execution and goes into a state in which the function body is executed. When execution of the function body is finished, the result is plugged back into the whole expression.

  This way, we correctly deal with non-terminating functions, and do not allow function calls to be interleaved [ISO12, 6.5.2.2p10].
- **Subexpressions that have undefined behavior.** A subexpression has undefined behavior if it has the shape of a head redex, but cannot be head reduced. In this case, the whole program semantics goes into a special $\overline{\mathsf{undef}}$ state that accounts for undefined behavior.

Formally, a subexpression has undefined behavior if it is an *unsafe redex*, which is captured by the following definitions.

Definition 6.4.6. *An expression is a* redex *if it has one of the following shapes:*

$$\begin{array}{lllll}
x_i & *[\nu]_\Omega & \&[\nu]_\Omega & [\nu]_\Omega \mathbin{.\mathbf{l}} r & [\nu]_\Omega \mathbin{.\mathbf{r}} r \\
[\nu_1]_{\Omega_1}[\vec{r} := [\nu_2]_{\Omega_2}] & [\nu]_{\Omega_1}\ \alpha\ [\nu]_{\Omega_2} & [\nu]_\Omega([\vec{\nu}]_{\vec{\Omega}}) & \mathsf{abort}\ \tau & \mathsf{load}\ [\nu]_\Omega \\
\mathsf{alloc}_\tau\ [\nu]_\Omega & \mathsf{free}\ [\nu]_\Omega & \circledcirc_u\ [\nu]_\Omega & [\nu_1]_{\Omega_1} \circledcirc [\nu_2]_{\Omega_2} & (\tau)[\nu]_\Omega \\
[\nu]_\Omega \mathrel{?} e_2 : e_3 & ([\nu]_\Omega, e) & & &
\end{array}$$

Definition 6.4.7. *An expression e is* safe*, notation* $\Gamma, \rho \vdash (e, m)\ \mathsf{safe}$*, if either:*

[1] The comma operator `(e1,e2)` should not be confused with the commas that separate function arguments `f(e1,e2)`. The arguments of the comma operator are guaranteed to be evaluated left to right, and the value of the first operand is thrown away. Function arguments, on the other hand, may be evaluated in any order.

1. *The expression e is a function call. That is, $e = [\nu]_\Omega([\vec{\nu}]_{\vec{\Omega}})$.*
2. *There is an expression e' and memory m' such that $\Gamma, \rho \vdash (e, m) \to_{\mathsf{h}} (e', m')$.*

The $\mathsf{abort}\ \tau$ expression does not have a rule for head reduction, and is therefore an unsafe redex. This is intended because $\mathsf{abort}\ \tau$ is used to describe an expression that when reached has undefined behavior.

The expressions $e_1\ ?\ e_2 : e_3$ and (e_1, e_2) play an important role in the *sequence point restriction*, which states that an object may not be modified more than once, or being read after being modified, between two *sequence points* [ISO12, 6.5p2]. The expressions $e_1\ ?\ e_2 : e_3$ and (e_1, e_2) contain a sequence point. For example:

```
int x = 0, *p = &x;
x + x;               // OK
(x = 3) + (x = 4);   // Undefined, x modified twice
(x = 3) + (*p = 4);  // Undefined, x modified twice
(x = 3) + x;         // Undefined, x can be read after being modified
(x = 3, x = 4);      // OK, the , construct contains a sequence point
((x = 3) ? x : 1);   // OK, the ? construct contains a sequence point
```

To describe the sequence point restriction formally, we let rule 7 of Definition 6.4.1 for assignments not only store the value v at address a, but also let it lock a:

$$\frac{\mathsf{writable}_\Gamma\ a\ m \qquad \Gamma, m \vdash (a\ \alpha\ v) \mapsto_{\mathsf{ass}} (v_a,\ v')}{\Gamma, \delta \vdash ([a]_{\Omega_1}\ \alpha\ [v]_{\Omega_2}, m) \to_{\mathsf{h}} ([v']_{\{a\}_\Gamma \cup \Omega_1 \cup \Omega_2}, \mathsf{lock}_\Gamma\ a\ (m\langle a := v_a\rangle_\Gamma))}$$

By locking a, we temporarily reduce its permissions in order to make consecutive accesses to it illegal. We furthermore add a to the set $\{a\}_\Gamma \cup \Omega_1 \cup \Omega_2$ to keep track of the fact it has been locked. Rule 14 for the comma $([v]_\Omega, e)$ and rules 15 and 16 for the conditional $[v]_\Omega\ ?\ e_2 : e_3$ describe a sequence point by unlocking the set Ω in memory, which makes future accesses to Ω possible again. Let us consider an example:

$$\begin{tikzcd}
 & ((a_1 := 3) + (a_2 := 4), e) \arrow[dl, "\text{lock } a_1"'] \arrow[dr, "\text{lock } a_2"] & \\
([3]_{\{a_1\}} + (a_2 := 4), e) \arrow[dr, "\text{lock } a_2"'] & & ((a_1 := 3) + [4]_{\{a_2\}}, e) \arrow[dl, "\text{lock } a_1"] \\
 & ([7]_{\{a_1, a_2\}}, e) \arrow[d, "\text{unlock } \{a_1, a_2\}"] & \\
 & e &
\end{tikzcd}$$

In case the addresses a_1 and a_2 are the same or do partially overlap, the assignment executed last has undefined behavior. That is because the first assignment locks the address, which in turn causes the $\mathsf{writable}_\Gamma$ condition of rule 7 to fail. If a_1 and a_2 do not overlap, no undefined behavior due to sequence point violations occurs. Notice that accesses to a_1 or a_2 in the second operand of the comma operator are legal as its corresponding sequence point unlocks $\{a_1, a_2\}$.

As discussed in Section 2.5.9, there is a sequence point before each function call. The following program thus non-deterministically returns 3 or 4:

```
int assign(int *p, int y) { return *p = y; }
int main() {
  int x;
  assign(&x, 3) + assign(&x, 4);
  return x;
}
```

Sequence points before function calls, as well as sequence points as part of statements such as $\mathsf{if}\ (e)\ s_1\ \mathsf{else}\ s_2$, are described in Section 6.5.

Our operational semantics of expressions is influenced by Norrish [Nor98] who also uses evaluation contexts to describe non-determinism in C expressions. Evaluation contexts themselves were introduced by Felleisen *et al.* [FFKD87] in the context of λ-calculus with control operators. In order to describe the sequence point restriction, Norrish also keeps track of memory areas that have been modified. However, he did not integrate locks into a more general permission system.

Similar to Norrish's C++ semantics [Nor08] and Ellison and Roşu [ER12b], we implicitly use non-determinism to describe all undefined behaviors caused by sequence point violations. For example, in `x + (x = 10)` only one execution order (executing the read after the assignment) leads to a sequence point violation. Both execution orders lead to a sequence point violation in Norrish's C semantics [Nor98] because he also keeps track of a set of memory locations that have been read.

We assign undefined behavior to more sequence point violations than Ellison and Roşu [ER12b] and the C11 standard [ISO12, 5.1.2.3] do. Ellison and Roşu release the locks of *all* objects at each sequence point, whereas we just release the locks that have been created by the subexpression in which the sequence point occurs. For example, we assign undefined behavior to the following program:

```
int assign(int *p, int y) { return *p = y; }
int main() {
  int x;
  (x = 3) + assign(&x, 4);
  return x;
}
```

The execution order that leads to undefined behavior is: (a) `x = 3`, which locks `x` (b) call `assign(&x, 4)`, which releases just the locks that have been created by the subexpressions `&x` and `4` (*i.e.* the empty set) (c) the assignment `*p = y`, in which `p` points to the locked object `x`. Ellison and Roşu assign defined behavior to the above program. They let the sequence point before `assign(&x, 4)` release all locks, including the lock of `x` that belongs to another subexpression.

We believe that our treatment of the sequence point restriction has some advantages over the treatment by the C11 standard and that by Ellison and Roşu:

- Our treatment of the sequence point restriction gives rise to strictly more undefined behavior, but only in artificial corner cases. More undefined behavior allows for more effective optimizations by compilers.
- Dealing with sequence points *locally* instead of *globally* provides a more convenient metatheory. In particular, it brings the operational semantics closer to separation logic because separation logic operates on local parts of the memory instead of the whole memory (see Section 8.4).

We believe that only artificial programs become illegal because different function calls in the same expression are still allowed to write to a shared part of the memory (which is useful for memoization). For example:

```
int f(int y) {
  static int map[MAP_SIZE];
  if (map[y]) { return map[y]; }
  return (map[y] = expensive_function(y));
}
```

In the context of this example, the expression `f(3) + f(3)` has still defined behavior according to CH$_2$O semantics.

An optimization that may be allowed by our treatment of sequence points, but that is not allowed by the C11 standard nor by Ellison and Roşu, is *by reference passing of struct and union values through expressions.* CompCert passes struct and union values through expressions by reference, and even describes it as such in its semantics. Only at assignments and function calls, struct and union values are actually copied. Let us consider an example:

```
struct S { int x; } s1 = { 1 }, s2 = { 2 };
int f() { if (s1.x == 2) { s2.x = 40; } return 0; }
int g(struct S s, int z) { return s.x; }
int main() { return g(s1 = s2, f()); }
```

This code has different behaviors in CompCert, the semantics of Ellison and Roşu, and the CH$_2$O semantics, namely:

- It is guaranteed to return $\{2, 40\}$ according to the non-deterministic semantics of CompCert C.
- It returns 40 when compiled with CompCert.
- It is guaranteed to return $\{2\}$ according to the semantics of Ellison and Roşu.
- It has undefined behavior in the CH$_2$O semantics.

Let us analyze the execution order by which the program yields `40` in CompCert. We start with the body of the function `main`:

1. The initial expression in the initial memory:
   `g(s1 = s2, f())` in `s1` $\mapsto \{1\}$, `s2` $\mapsto \{2\}$.
2. After the assignment `s1 = s2` is executed:
   `g(s2, f())` in `s1` $\mapsto \{2\}$, `s2` $\mapsto \{2\}$.

3. After the function `f` is called, which modifies `s2` because `s1.x == 2`: `g(s2,0)` in `s1` $\mapsto \{2\}$, `s2` $\mapsto \{40\}$.
4. After the function `f` is called, which returns the field `x` of the struct, the return values of `main` becomes `40`.

Step (2) is problematic because `s1 = s2` yields a reference to `s2` instead of the actual value `{struct S){ 2 }` of `s2`. Step (3) changes the value of `s2`, and step (4) thus uses the updated value of the struct. Step (3) has undefined behavior because it attempts to access `s1` which has been locked in step (2).

It would be interesting to investigate whether our semantics of the sequence point restriction justifies by reference passing of struct and union values through expressions in general. We conjecture so because by reference passing of struct an union values relies of being able to delay computations. Delaying of computations is exactly what the sequence point restriction in combination with non-deterministic expression execution should allow. However, if the sequence point before a function call releases all locks, fewer computations can be delayed.

6.5 Operational semantics

This section defines the reduction $S_1 \rightarrowtriangle S_2$ on program states. Program states are tuples $\mathbf{S}(\mathcal{P},\ \phi,\ m)$ where m is the memory and $(\mathcal{P},\ \phi)$ is a *zipper*-like data structure that describes the part of the program that is being executed. Execution of statements is modeled by traversal through the zipper in the directions *down* $\searrow$, *up* $\nearrow$, *goto* $\curvearrowright$, *throw* $\uparrow$ or *return* $\upuparrows$ corresponding to the current form of (non-local) control. Let us reconsider the example from Section 2.5.1:

```
int *p = NULL;
l: if (p) {
  return (*p);
} else {
  int j = 10; p = &j; goto l;
}
```

Figure 6.1 displays a part of the reduction sequence corresponding to the execution of this program. The sequence starts after the assignment `p = &j` has been performed. When the goto l statement is reached, the direction is changed to $\curvearrowright l$ in step $S_2 \rightarrowtriangle S_3$, which invokes a subsequent small-step traversal to search for the label l. During this traversal, the variable `j` is deallocated in step $S_5 \rightarrowtriangle S_6$.

The pair $(\mathcal{P},\ \phi)$, which is part of a program state $\mathbf{S}(\mathcal{P},\ \phi,\ m)$, is not an ordinary zipper. We adapt Huet's zipper data structure [Hue97] by annotating each block scope local variable in the context $\mathcal{P}$ with its associated memory index (see the annotation o in the $\mathsf{local}_{o:\tau}\ \Box$ construct in the example), and we also let $\mathcal{P}$ keep track of the full control stack (*i.e.* which functions have been called). Our adaptation of the zipper can also be seen as a generalization of continuations, as for example being used in CompCert [AB07, Ler09a]. However, there are some notable differences:

$$\begin{aligned}
\mathcal{P}_1 = {} & (\square\,;\mathsf{goto}\ l)\\
& (x_0 := \mathsf{int}_{\mathsf{signed\ int}}\, 10\,;\square)\\
& (\mathsf{local}_{o_j : \mathsf{signed\ int}}\ \square)\\
& (\mathsf{if}\ (\mathsf{load}\ x_0)\\
& \quad \mathsf{return}\ (\mathsf{load}\ (\&(\mathsf{load}\ x_0)))\\
& \ \mathsf{else}\ \square)\\
& (l : ;\square)\\
& (x_0 :=\\
& \quad \mathsf{ptr}\,(\mathsf{NULL\ signed\ int})\,;\square)\\
& (\mathsf{local}_{o_p : \mathsf{signed\ int}*}\ \square)\\
\phi_1 = {} & (\nearrow,\ x_1 := x_0)\\
S_1 = {} & \mathbf{S}(\mathcal{P}_1,\ \phi_1,\ m)
\end{aligned}$$

$$\begin{aligned}
\mathcal{P}_2 = {} & (x_1 := x_0\,;\square)\\
& (x_0 := \mathsf{int}_{\mathsf{signed\ int}}\, 10\,;\square)\\
& (\mathsf{local}_{o_j : \mathsf{signed\ int}}\ \square)\\
& (\mathsf{if}\ (\mathsf{load}\ x_0)\\
& \quad \mathsf{return}\ (\mathsf{load}\ (\&(\mathsf{load}\ x_0)))\\
& \ \mathsf{else}\ \square)\\
& (l : ;\square)\\
& (x_0 :=\\
& \quad \mathsf{ptr}\,(\mathsf{NULL\ signed\ int})\,;\square)\\
& (\mathsf{local}_{o_p : \mathsf{signed\ int}*}\ \square)\\
\phi_2 = {} & (\searrow,\ \mathsf{goto}\ l)\\
S_2 = {} & \mathbf{S}(\mathcal{P}_2,\ \phi_2,\ m)
\end{aligned}$$

$$\begin{aligned}
\mathcal{P}_3 = {} & (x_1 := x_0\,;\square)\\
& (x_0 := \mathsf{int}_{\mathsf{signed\ int}}\, 10\,;\square)\\
& (\mathsf{local}_{o_j : \mathsf{signed\ int}}\ \square)\\
& (\mathsf{if}\ (\mathsf{load}\ x_0)\\
& \quad \mathsf{return}\ (\mathsf{load}\ (\&(\mathsf{load}\ x_0)))\\
& \ \mathsf{else}\ \square)\\
& (l : ;\square)\\
& (x_0 :=\\
& \quad \mathsf{ptr}\,(\mathsf{NULL\ signed\ int})\,;\square)\\
& (\mathsf{local}_{o_p : \mathsf{signed\ int}*}\ \square)\\
\phi_3 = {} & (\curvearrowright l,\ \mathsf{goto}\ l)\\
S_3 = {} & \mathbf{S}(\mathcal{P}_3,\ \phi_3,\ m)
\end{aligned}$$

$$\begin{aligned}
\mathcal{P}_4 = {} & (x_0 := \mathsf{int}_{\mathsf{signed\ int}}\, 10\,;\square)\\
& (\mathsf{local}_{o_j : \mathsf{signed\ int}}\ \square)\\
& (\mathsf{if}\ (\mathsf{load}\ x_0)\\
& \quad \mathsf{return}\ (\mathsf{load}\ (\&(\mathsf{load}\ x_0)))\\
& \ \mathsf{else}\ \square)\\
& (l : ;\square)\\
& (x_0 :=\\
& \quad \mathsf{ptr}\,(\mathsf{NULL\ signed\ int})\,;\square)\\
& (\mathsf{local}_{o_p : \mathsf{signed\ int}*}\ \square)\\
\phi_4 = {} & (\curvearrowright l,\ x_1 := x_0\ ;\ \mathsf{goto}\ l)\\
S_4 = {} & \mathbf{S}(\mathcal{P}_4,\ \phi_4,\ m)
\end{aligned}$$

$$\begin{aligned}
\mathcal{P}_5 = {} & (\mathsf{local}_{o_j : \mathsf{signed\ int}}\ \square)\\
& (\mathsf{if}\ (\mathsf{load}\ x_0)\\
& \quad \mathsf{return}\ (\mathsf{load}\ (\&(\mathsf{load}\ x_0)))\\
& \ \mathsf{else}\ \square)\\
& (l : ;\square)\\
& (x_0 :=\\
& \quad \mathsf{ptr}\,(\mathsf{NULL\ signed\ int})\,;\square)\\
& (\mathsf{local}_{o_p : \mathsf{signed\ int}*}\ \square)\\
\phi_5 = {} & (\curvearrowright l,\ x_0 := \mathsf{int}_{\mathsf{signed\ int}}\, 10\ ;\\
& \qquad x_1 := x_0\ ;\ \mathsf{goto}\ l)\\
S_5 = {} & \mathbf{S}(\mathcal{P}_5,\ \phi_5,\ m)
\end{aligned}$$

$$\begin{aligned}
\mathcal{P}_6 = {} & (\mathsf{if}\ (\mathsf{load}\ x_0)\\
& \quad \mathsf{return}\ (\mathsf{load}\ (\&(\mathsf{load}\ x_0)))\\
& \ \mathsf{else}\ \square)\\
& (l : ;\square)\\
& (x_0 :=\\
& \quad \mathsf{ptr}\,(\mathsf{NULL\ signed\ int})\,;\square)\\
& (\mathsf{local}_{o_p : \mathsf{signed\ int}*}\ \square)\\
\phi_6 = {} & (\curvearrowright l,\ \mathsf{local}_{\mathsf{signed\ int}}\,(\\
& \qquad x_0 := \mathsf{int}_{\mathsf{signed\ int}}\, 10\ ;\\
& \qquad x_1 := x_0\ ;\ \mathsf{goto}\ l))\\
S_6 = {} & \mathbf{S}(\mathcal{P}_6,\ \phi_6,\ \mathsf{free}\ o_j\ m)
\end{aligned}$$

Figure 6.1: An example reduction. The reduction $S_1 \twoheadrightarrow S_2$ is by rule 4b, $S_2 \twoheadrightarrow^* S_5$ by rule 6c and $S_5 \twoheadrightarrow S_6$ by rule 7b of Definition 6.5.8.

- Our zipper implicitly contains the local stack, which assigns memory indices to local variables, see Definition 6.1.10, whereas a semantics based on continuation typically stores the local stack separately.
- Our zipper preserves the part of the statement that already has been executed, whereas a continuation only contains the part that remains to be done.
- Since the complete program is preserved, looping constructs such as our loop statement do not have to duplicate code.

In order to describe the interaction between non-local control flow and block scope variables, it is essential that we not only keep track of the part of the statement that remains to be executed, but also preserve the part that has been executed already. Occurrences of non-local control flow invoke a small-step traversal through the zipper during which we can perform the required allocations and deallocations in a natural way. The point of this traversal is thus not so much to *search* for the label, but to incrementally *calculate* the required allocations and deallocations.

Definition 6.5.1. Statement contexts *are inductively defined as:*

$$\mathcal{S}_{\mathsf{s}} \in \mathsf{sctx}_{\mathsf{s}} ::= \mathsf{catch}\ \square \mid \square\,; s_2 \mid s_1\,; \square \mid \mathsf{loop}\ \square \mid \mathsf{if}\ (e)\ \square\ \mathsf{else}\ s_2 \mid \mathsf{if}\ (e)\ s_1\ \mathsf{else}\ \square$$

Given a statement context $\mathcal{S}_{\mathsf{s}}$ *and a statement* s*,* substitution of s for the hole $\square$ in $\mathcal{S}_{\mathsf{s}}$*, notation* $\mathcal{S}_{\mathsf{s}}[\,s\,]$*, is defined as usual.*

Definition 6.5.2. *The judgment* $\Gamma, \Delta, \vec{\tau} \vdash \mathcal{S}_{\mathsf{s}} : (\alpha,\, \tau^?) \rightarrowtail (\beta,\, \sigma^?)$ *describes that* $\mathcal{S}_{\mathsf{s}}$ is a valid statement context of type $\tau^?$ with a hole $\square$ of type $\sigma^?$. *It is inductively defined as:*

$$\frac{}{\Gamma, \Delta, \vec{\tau} \vdash \mathsf{catch}\ \square : (\beta,\, \sigma^?) \rightarrowtail (\mathsf{false},\, \sigma^?)} \qquad \frac{\Gamma, \Delta, \vec{\tau} \vdash s_2 : (\beta_2,\, \tau_2^?) \qquad \tau_1^?; \tau_2^? \succ \sigma^?}{\Gamma, \Delta, \vec{\tau} \vdash \square\, ;\, s_2 : (\beta_2,\, \tau_1^?) \rightarrowtail (\beta_2,\, \sigma^?)}$$

$$\frac{\Gamma, \Delta, \vec{\tau} \vdash s_1 : (\beta_1,\, \tau_1^?) \qquad \tau_1^?; \tau_2^? \succ \sigma^?}{\Gamma, \Delta, \vec{\tau} \vdash s_1\, ;\, \square : (\beta_2,\, \tau_2^?) \rightarrowtail (\beta_2,\, \sigma^?)} \qquad \frac{}{\Gamma, \Delta, \vec{\tau} \vdash \mathsf{loop}\ \square : (\beta,\, \sigma^?) \rightarrowtail (\mathsf{true},\, \sigma^?)}$$

$$\frac{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e : \tau_{\mathsf{b}} \qquad \mathsf{locks}\ e = \emptyset \qquad \Gamma, \Delta, \vec{\tau} \vdash s_2 : (\beta_2,\, \tau_2^?) \qquad \tau_1^?; \tau_2^? \succ \sigma^?}{\Gamma, \Delta, \vec{\tau} \vdash \mathsf{if}\ (e)\ \square\ \mathsf{else}\ s_2 : (\beta_1,\, \tau_1^?) \rightarrowtail (\beta_1 \wedge \beta_2,\, \sigma^?)}$$

$$\frac{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e : \tau_{\mathsf{b}} \qquad \tau_{\mathsf{b}} \neq \mathsf{void} \qquad \mathsf{locks}\ e = \emptyset \qquad \Gamma, \Delta, \vec{\tau} \vdash s_1 : (\beta_1,\, \tau_1^?) \qquad \tau_1^?; \tau_2^? \succ \sigma^?}{\Gamma, \Delta, \vec{\tau} \vdash \mathsf{if}\ (e)\ s_1\ \mathsf{else}\ \square : (\beta_2,\, \tau_2^?) \rightarrowtail (\beta_1 \wedge \beta_2,\, \sigma^?)}$$

A pair $(\vec{\mathcal{S}_{\mathsf{s}}},\, s)$ of a list of statement contexts $\vec{\mathcal{S}_{\mathsf{s}}}$ and a *focused* substatement s forms a zipper for statements without block scope variables. That means, $\vec{\mathcal{S}_{\mathsf{s}}}$ is a statement turned inside-out that represents a path from the focused substatement s to the top of the whole statement. In the subsequent definitions we will extend the context $\vec{\mathcal{S}_{\mathsf{s}}}$ to account for the local stack as well as the whole call stack.

Definition 6.5.3. Expression statement contexts *are inductively defined as:*

$$\mathcal{S}_{\mathsf{e}} \in \mathsf{sctx}_{\mathsf{e}} ::= \square \mid \mathsf{return}\ \square \mid \mathsf{if}\ (\square)\ s_1\ \mathsf{else}\ s_2$$

Given an expression statement context $\mathcal{S}_{\mathsf{e}}$ *and an expression* e*,* substitution of e for the hole $\square$ in $\mathcal{S}_{\mathsf{e}}$*, notation* $\mathcal{S}_{\mathsf{e}}[\, e\,]$*, is defined as usual.*

Definition 6.5.4. *The judgment* $\Gamma, \Delta, \vec{\tau} \vdash \mathcal{S}_{\mathsf{e}} : \tau \rightarrowtail (\beta,\, \sigma^?)$ *describes that* $\mathcal{S}_{\mathsf{e}}$ is a valid expression statement context of type $(\beta,\, \sigma^?)$ with a hole $\square$ of type τ. *It is inductively defined as:*

$$\frac{}{\Gamma, \Delta, \vec{\tau} \vdash \square : \tau \rightarrowtail (\mathsf{false},\, \bot)} \qquad \frac{}{\Gamma, \Delta, \vec{\tau} \vdash \mathsf{return}\ \square : \tau \rightarrowtail (\mathsf{true},\, \tau)}$$

$$\frac{\tau_{\mathsf{b}} \neq \mathsf{void} \qquad \Gamma, \Delta, \vec{\tau} \vdash s_1 : (\beta_1,\, \tau_1^?) \qquad \Gamma, \Delta, \vec{\tau} \vdash s_2 : (\beta_2,\, \tau_2^?) \qquad \tau_1^?; \tau_2^? \succ \sigma^?}{\Gamma, \Delta, \vec{\tau} \vdash \mathsf{if}\ (\square)\ s_1\ \mathsf{else}\ s_2 : \tau_{\mathsf{b}} \rightarrowtail (\beta_1 \wedge \beta_2,\, \sigma^?)}$$

Definition 6.5.5. Program contexts, directions, focuses *and* program states *are inductively defined as:*

$$\begin{aligned}
\mathcal{P}_{\mathsf{s}} \in \mathsf{ctx}_{\mathsf{s}} &::= \mathcal{S}_{\mathsf{s}} \mid \mathsf{local}_{o:\tau}\ \square \mid (\mathcal{S}_{\mathsf{e}}, e) \mid \mathsf{resume}\ \mathcal{E} \mid \mathsf{params}\ f\ \overrightarrow{o\tau} \\
\mathcal{P} \in \mathsf{ctx} &:= \mathsf{list}\ \mathsf{ctx}_{\mathsf{s}} \\
d \in \mathsf{direction} &::= \searrow \mid \nearrow \mid \curvearrowright l \mid \uparrow n \mid \upuparrows v \\
\phi_U \in \mathsf{undef} &::= \lightning\, \mathcal{E}\langle e \rangle \mid \lightning\, \mathcal{S}_{\mathsf{e}}\langle [v]_\Omega \rangle \\
\phi \in \mathsf{focus} &::= (d,\, s) \mid e \mid \overline{\mathsf{call}}\ f\ \vec{v} \mid \overline{\mathsf{return}}\ f\ v \mid \overline{\mathsf{undef}}\ \phi_U \\
S \in \mathsf{state} &::= \mathbf{S}(\mathcal{P},\, \phi,\, m)
\end{aligned}$$

The projection $\mathsf{mem} : \mathsf{state} \to \mathsf{mem}$ *is defined as expected.*

Program states $\mathbf{S}(\mathcal{P},\, \phi,\, m)$ contain a zipper $(\mathcal{P},\, \phi)$ where the context $\mathcal{P}$ describes the location of the focused part ϕ in the program that is being executed. The essential difference with a traditional zipper is that each $\mathsf{local}_{\tau:o}\,\Box$ construct is annotated with its block scope variable. The construct $\mathsf{resume}\ \mathcal{E}$ delimits a callee from its caller, and the construct $\mathsf{params}\ f\ \vec{o\tau}$ associate a list $\vec{o}$ of memory indices to the parameters of a function. Since a program context thereby implicitly contains the stack, we define an erasure function to extract the stack from it.

Definition 6.5.6. *The function* $\mathsf{locals} : \mathsf{ctx} \to \mathsf{stack}$ *yields the* corresponding local stack *of a program context. It is defined as:*

$$\begin{aligned}
\mathsf{locals}\ \varepsilon &:= \varepsilon \\
\mathsf{locals}\ (\mathcal{S}_{\mathsf{s}}\,\mathcal{P}) &:= \mathsf{locals}\ \mathcal{P} \\
\mathsf{locals}\ ((\mathsf{local}_{o:\tau}\,\Box)\,\mathcal{P}) &:= (o,\, \tau)\,(\mathsf{locals}\ \mathcal{P}) \\
\mathsf{locals}\ ((\mathcal{S}_{\mathsf{e}}, e)\,\mathcal{P}) &:= \mathsf{locals}\ \mathcal{P} \\
\mathsf{locals}\ ((\mathsf{resume}\ \mathcal{E})\,\mathcal{P}) &:= \varepsilon \\
\mathsf{locals}\ ((\mathsf{params}\ f\ \vec{o\tau})\,\mathcal{P}) &:= \vec{o\tau}\,(\mathsf{locals}\ \mathcal{P})
\end{aligned}$$

We define $\mathsf{locals}\ (\mathsf{resume}\ \mathcal{E}\,\mathcal{P})$ as ε instead of $\mathsf{locals}\ \mathcal{P}$ to ensure that a callee cannot refer to the local variables of its caller.

Before we will consider the actual definition of the reduction relation $S_1 \rightarrowtriangle S_2$ in Definition 6.5.8), we describe the different forms of program execution by representative rules.

- **Statements.** The state $\mathbf{S}(\mathcal{P},\, (d,\, s),\, m)$ describes execution of a statement s in direction d. The direction describes the current form of (non-local) control. For example, when a $\mathsf{goto}\ l$ statement is reached, the direction is changed into $\curvearrowright l$, and a small-step traversal through the zipper is performed until the label l has been reached where normal control flow is resumed. This is described by the following rules:

$$\begin{aligned}
\mathbf{S}(\mathcal{P},\, (\searrow,\, \mathsf{goto}\ l),\, m) &\rightarrowtriangle \mathbf{S}(\mathcal{P},\, (\curvearrowright l,\, \mathsf{goto}\ l),\, m) \\
\mathbf{S}(\mathcal{P},\, (\curvearrowright l,\, \mathcal{S}_{\mathsf{s}}[\,s\,]),\, m) &\rightarrowtriangle \mathbf{S}(\mathcal{S}_{\mathsf{s}}\,\mathcal{P},\, (\curvearrowright l,\, s),\, m) && \text{if } l \in \mathsf{labels}\ s \\
\mathbf{S}(\mathcal{S}_{\mathsf{s}}\,\mathcal{P},\, (\curvearrowright l,\, s),\, m) &\rightarrowtriangle \mathbf{S}(\mathcal{P},\, (\curvearrowright l,\, \mathcal{S}_{\mathsf{s}}[\,s\,]),\, m) && \text{if } l \notin \mathsf{labels}\ s \\
\mathbf{S}(\mathcal{P},\, (\curvearrowright l,\, \mathsf{local}_{\tau}\ s),\, m) &\rightarrowtriangle \mathbf{S}((\mathsf{local}_{\tau:o}\,\Box)\,\mathcal{P},\, (\curvearrowright l,\, s),\, \mathsf{alloc}_{\Gamma}\ o\ (\mathsf{new}_{\Gamma}\ \tau)\ \mathsf{false}\ m) \\
& \qquad\qquad\qquad\qquad \text{if } o \notin \mathsf{dom}\ m \text{ and } l \in \mathsf{labels}\ s \\
\mathbf{S}((\mathsf{local}_{o:\tau}\,\Box)\,\mathcal{P},\, (\curvearrowright l,\, s),\, m) &\rightarrowtriangle \mathbf{S}(\mathcal{P},\, (\curvearrowright l,\, \mathsf{local}_{\tau}\ s),\, \mathsf{free}\ o\ m) && \text{if } l \notin \mathsf{labels}\ s \\
\mathbf{S}(\mathcal{P},\, (\curvearrowright l,\, l :),\, m) &\rightarrowtriangle \mathbf{S}(\mathcal{P},\, (\nearrow,\, l :),\, m)
\end{aligned}$$

  The construct $\mathsf{local}_{o:\tau}\,\Box$ associates the block scope variable with an object identifier o that refers its object in memory. Local variables are initialized with the indeterminate value $\mathsf{new}_{\Gamma}\ \tau$ (Definition 5.5.14 on page 84).

- **Undefined behavior.** The state $\mathbf{S}(\mathcal{P},\, \overline{\mathsf{undef}}\,\phi_U,\, m)$ describes an occurrence of undefined behavior. The annotation ϕ_U indicates the source of the undefined behavior.

- **Expressions.** The state $\mathbf{S}(\mathcal{P},\, e,\, m)$ describes execution of an expression e. Most notably, provided $\Gamma, \mathsf{locals}\ \mathcal{P} \vdash (e_1, m_1) \rightarrowtriangle_{\mathsf{h}} (e_2, m_2)$, we have:

  $$\mathbf{S}(\mathcal{P},\, \mathcal{E}[\,e_1\,],\, m_1) \rightarrowtriangle \mathbf{S}(\mathcal{P},\, \mathcal{E}[\,e_2\,],\, m_2).$$

  Non-deterministic decomposition of an expression into an expression context $\mathcal{E}$ and subexpression e_1 models the unspecified order of expressions evaluation.

  If the expression e_1 is an unsafe redex (a redex that cannot be $\rightarrowtriangle_{\mathsf{h}}$-reduced due to undefined behavior, see Definitions 6.4.6 and 6.4.7), we have:

  $$\mathbf{S}(\mathcal{P},\, \mathcal{E}[\,e_1\,],\, m) \rightarrowtriangle \mathbf{S}(\mathcal{P},\, \overline{\mathsf{undef}}\,(\text{↯}\,\mathcal{E}\langle e_1\rangle),\, m).$$

- **Function calls.** The state $\mathbf{S}(\mathcal{P},\, \overline{\mathsf{call}}\, f\, \vec{v},\, m)$ describes calling a function f with arguments $\vec{v}$. For function calls we have the following rules:

  $$\begin{aligned}\mathbf{S}(\mathcal{P},\, \mathcal{E}[\,[f^{\vec{\tau}\mapsto\tau}]_{\Omega}([\vec{v}]_{\vec{\Omega}})\,],\, m) &\rightarrowtriangle \mathbf{S}((\mathsf{resume}\ \mathcal{E})\,\mathcal{P},\, \overline{\mathsf{call}}\, f\, \vec{v},\, \mathsf{unlock}\ (\Omega \cup \bigcup \vec{\Omega})\ m)\\ \mathbf{S}(\mathcal{P},\, \overline{\mathsf{call}}\, f\, \vec{v},\, m) &\rightarrowtriangle \mathbf{S}((\mathsf{params}\ f\ \overrightarrow{o\tau})\,\mathcal{P},\, (\searrow,\, s),\, \mathsf{alloc}_{\Gamma}\, \vec{o}\, \vec{v}\ \mathsf{false}\ m)\end{aligned}$$

  The construct $\mathsf{resume}\ \mathcal{E}$ keeps track of the expression context of the caller and delimits the local stack of the callee from the one of the caller.

  The second rule allocates the function arguments $\vec{v}$ of type $\vec{\tau}$ in memory. The construct $\mathsf{params}\ f\ \overrightarrow{o\tau}$ associates the object identifiers $\vec{o}$ to the function arguments. The statement s is the body of f, that is $\delta\, f = s$.

  When a function f returns with value v, the control is given back to the caller, which continue execution of the expression $\mathcal{E}[\,[v]_{\emptyset}\,]$. Returning from a function involves the intermediate state $\mathbf{S}(\mathcal{P},\, \overline{\mathsf{return}}\, f\, v,\, m)$. The corresponding rules are as follows:

  $$\begin{aligned}\mathbf{S}((\mathsf{params}\ f\ \overrightarrow{o\tau})\,\mathcal{P},\, (\upuparrows v,\, s),\, m) &\rightarrowtriangle \mathbf{S}(\mathcal{P},\, \overline{\mathsf{return}}\, f\, v,\, \mathsf{free}\ \vec{o}\, m)\\ \mathbf{S}((\mathsf{resume}\ \mathcal{E})\,\mathcal{P},\, \overline{\mathsf{return}}\, f\, v,\, m) &\rightarrowtriangle \mathbf{S}(\mathcal{P},\, \mathcal{E}[\,[v]_{\emptyset}\,],\, m)\end{aligned}$$

We have adapted the approach of distinguishing different kinds of program states that correspond to different kinds of program execution from Leroy [Ler09a]. Our treatment of statement execution is entirely different from Leroy's.

Definition 6.5.7. *The judgments* $(d,\, s)\ \mathsf{in}$ *and* $(d,\, s)\ \mathsf{out}$ *denote whether the direction* d *is* inwards *or* outwards *in the statement* s*. These are inductively defined as:*

$$\frac{}{(\searrow,\, s)\ \mathsf{in}} \qquad \frac{l \in \mathsf{labels}\ s}{(\curvearrowright l,\, s)\ \mathsf{in}} \qquad \frac{}{(\nearrow,\, s)\ \mathsf{out}} \qquad \frac{l \notin \mathsf{labels}\ s}{(\curvearrowright l,\, s)\ \mathsf{out}} \qquad \frac{}{(\uparrow n,\, s)\ \mathsf{out}} \qquad \frac{}{(\upuparrows v,\, s)\ \mathsf{out}}$$

Definition 6.5.8. *The* small-step reduction $\Gamma, \delta \vdash S_1 \rightarrowtriangle S_2$ *is inductively defined by the following rules:*

1. For simple statements:
   a) $\mathbf{S}(\mathcal{P},\, (\searrow,\, \mathsf{skip}),\, m) \rightarrowtriangle \mathbf{S}(\mathcal{P},\, (\nearrow,\, \mathsf{skip}),\, m)$
   b) $\mathbf{S}(\mathcal{P},\, (\searrow,\, \mathsf{goto}\ l),\, m) \rightarrowtriangle \mathbf{S}(\mathcal{P},\, (\curvearrowright l,\, \mathsf{goto}\ l),\, m)$

c) $\mathbf{S}(\mathcal{P},\ (\searrow,\ \mathsf{throw}\ n),\ m) \rightarrow \mathbf{S}(\mathcal{P},\ (\uparrow n,\ \mathsf{throw}\ n),\ m)$

d) $\mathbf{S}(\mathcal{P},\ (\searrow,\ l\ :),\ m) \rightarrow \mathbf{S}(\mathcal{P},\ (\nearrow,\ l\ :),\ m)$

e) $\mathbf{S}(\mathcal{P},\ (\searrow,\ \mathcal{S}_{\mathsf{e}}[\,e\,]),\ m) \rightarrow \mathbf{S}((\mathcal{S}_{\mathsf{e}}, e)\,\mathcal{P},\ e,\ m)$

2. For expressions:

a) $\mathbf{S}(\mathcal{P},\ \mathcal{E}[\,e_1\,],\ m_1) \rightarrow \mathbf{S}(\mathcal{P},\ \mathcal{E}[\,e_2\,],\ m_2)$
for any e_2 *and* m_2 *with* $\Gamma, \mathsf{locals}\ \mathcal{P} \vdash (e_1, m_1) \rightarrow_{\mathsf{h}} (e_2, m_2)$

b) $\mathbf{S}(\mathcal{P},\ \mathcal{E}[\,[f^{\vec{\tau}\mapsto\tau}]_{\Omega}([\vec{v}]_{\vec{\Omega}})\,],\ m) \rightarrow \mathbf{S}(\mathsf{resume}\ \mathcal{E}\,\mathcal{P},\ \overline{\mathsf{call}}\,f\,\vec{v},\ \mathsf{unlock}\ (\Omega \cup \bigcup \vec{\Omega})\ m)$

c) $\mathbf{S}(\mathcal{P},\ \mathcal{E}[\,e\,],\ m) \rightarrow \mathbf{S}(\mathcal{P},\ \overline{\mathsf{undef}}\ (\lightning\,\mathcal{E}\langle e\rangle),\ m)$
if e *is a redex with* $\Gamma, \mathsf{locals}\ \mathcal{P} \nvdash (m, e)\ \mathsf{safe}$

3. For finished expressions:

a) $\mathbf{S}((\square, e)\,\mathcal{P},\ [v]_{\Omega},\ m) \rightarrow \mathbf{S}(\mathcal{P},\ (\nearrow,\ e),\ \mathsf{unlock}\ \Omega\ m)$

b) $\mathbf{S}((\mathsf{return}\ \square, e)\,\mathcal{P},\ [v]_{\Omega},\ m) \rightarrow \mathbf{S}(\mathcal{P},\ (\upuparrows v,\ \mathsf{return}\ e),\ \mathsf{unlock}\ \Omega\ m)$

c) $\mathbf{S}((\mathsf{if}\ (\square)\ s_1\ \mathsf{else}\ s_2, e)\,\mathcal{P},\ [v_{\mathsf{b}}]_{\Omega},\ m) \rightarrow \mathbf{S}((\mathsf{if}\ (e)\ \square\ \mathsf{else}\ s_2)\,\mathcal{P},\ (\searrow,\ s_1),\ \mathsf{unlock}\ \Omega\ m)$
if $m \vdash (\mathsf{zero}\ v_{\mathsf{b}})\ \mathsf{defined}$ *and* $\neg\mathsf{zero}\ v_{\mathsf{b}}$

d) $\mathbf{S}((\mathsf{if}\ (\square)\ s_1\ \mathsf{else}\ s_2, e)\,\mathcal{P},\ [v_{\mathsf{b}}]_{\Omega},\ m) \rightarrow \mathbf{S}((\mathsf{if}\ (e)\ s_1\ \mathsf{else}\ \square)\,\mathcal{P},\ (\searrow,\ s_2),\ \mathsf{unlock}\ \Omega\ m)$
if $\mathsf{zero}\ v_{\mathsf{b}}$

e) $\mathbf{S}((\mathsf{if}\ (\square)\ s_1\ \mathsf{else}\ s_2, e)\,\mathcal{P},\ [v_{\mathsf{b}}]_{\Omega},\ m) \rightarrow \mathbf{S}(\mathcal{P},\ \overline{\mathsf{undef}}\ (\lightning\ (\mathsf{if}\ (\square)\ s_1\ \mathsf{else}\ s_2)\langle[v_{\mathsf{b}}]_{\Omega}\rangle),\ m)$
if $m \nvdash (\mathsf{zero}\ v_{\mathsf{b}})\ \mathsf{defined}$

4. For compound statements:

a) $\mathbf{S}(\mathcal{P},\ (\searrow,\ s_1\ ;\ s_2),\ m) \rightarrow \mathbf{S}((\square\,; s_2)\,\mathcal{P},\ (\searrow,\ s_1),\ m)$

b) $\mathbf{S}((\square\,; s_2)\,\mathcal{P},\ (\nearrow,\ s_1),\ m) \rightarrow \mathbf{S}((s_1\,; \square)\,\mathcal{P},\ (\searrow,\ s_2),\ m)$

c) $\mathbf{S}((s_1\,; \square)\,\mathcal{P},\ (\nearrow,\ s_2),\ m) \rightarrow \mathbf{S}(\mathcal{P},\ (\nearrow,\ s_1\ ;\ s_2),\ m)$

d) $\mathbf{S}(\mathcal{P},\ (\searrow,\ \mathsf{catch}\ s),\ m) \rightarrow \mathbf{S}((\mathsf{catch}\ \square)\,\mathcal{P},\ (\searrow,\ s),\ m)$

e) $\mathbf{S}((\mathsf{catch}\ \square)\,\mathcal{P},\ (\nearrow,\ s),\ m) \rightarrow \mathbf{S}(\mathcal{P},\ (\nearrow,\ \mathsf{catch}\ s),\ m)$

f) $\mathbf{S}(\mathcal{P},\ (\searrow,\ \mathsf{loop}\ s),\ m) \rightarrow \mathbf{S}((\mathsf{loop}\ \square)\,\mathcal{P},\ (\searrow,\ s),\ m)$

g) $\mathbf{S}((\mathsf{loop}\ \square)\,\mathcal{P},\ (\nearrow,\ s),\ m) \rightarrow \mathbf{S}(\mathcal{P},\ (\searrow,\ \mathsf{loop}\ s),\ m)$

h) $\mathbf{S}((\mathsf{if}\ (e)\ \square\ \mathsf{else}\ s_2)\,\mathcal{P},\ (\nearrow,\ s_1),\ m) \rightarrow \mathbf{S}(\mathcal{P},\ (\nearrow,\ \mathsf{if}\ (e)\ s_1\ \mathsf{else}\ s_2),\ m)$

i) $\mathbf{S}((\mathsf{if}\ (e)\ s_1\ \mathsf{else}\ \square)\,\mathcal{P},\ (\nearrow,\ s_2),\ m) \rightarrow \mathbf{S}(\mathcal{P},\ (\nearrow,\ \mathsf{if}\ (e)\ s_1\ \mathsf{else}\ s_2),\ m)$

5. For function calls:

a) $\mathbf{S}(\mathcal{P},\ \overline{\mathsf{call}}\,f\,\vec{v},\ m) \rightarrow \mathbf{S}((\mathsf{params}\ f\ \overrightarrow{o\tau})\,\mathcal{P},\ (\searrow,\ s),\ \mathsf{alloc}_{\Gamma}\ \vec{o}\ \vec{v}\ \mathsf{false}\ m)$
for the s *with* $\delta\, f = s$, *and any object identifiers* $\vec{o}$ *without duplicates and types* $\vec{\tau}$, *satisfying* $|\vec{o}| = |\vec{v}| = |\vec{\tau}|$, $\vec{o}_i \notin \mathsf{dom}\ m$ *and* $\tau_i = \mathsf{typeof}\ v_i$ *for each* $i < |\vec{v}|$

b) $\mathbf{S}((\mathsf{params}\ f\ \overrightarrow{o\tau})\,\mathcal{P},\ (\nearrow,\ s),\ m) \rightarrow \mathbf{S}(\mathcal{P},\ \overline{\mathsf{return}}\,f\ \mathsf{nothing},\ \mathsf{free}\ \vec{o}\ m)$

c) $\mathbf{S}((\mathsf{params}\ f\ \overrightarrow{o\tau})\,\mathcal{P},\ (\upuparrows v,\ s),\ m) \rightarrow \mathbf{S}(\mathcal{P},\ \overline{\mathsf{return}}\,f\ v,\ \mathsf{free}\ \vec{o}\ m)$

d) $\mathbf{S}((\mathsf{resume}\ \mathcal{E})\,\mathcal{P},\ \overline{\mathsf{return}}\,f\ v,\ m) \rightarrow \mathbf{S}(\mathcal{P},\ \mathcal{E}[\,[v]_{\emptyset}\,],\ m)$

6. For non-local control flow:

a) $\mathbf{S}(\mathcal{P},\ (\curvearrowright l,\ l\ :),\ m) \rightarrow \mathbf{S}(\mathcal{P},\ (\searrow,\ l\ :),\ m)$

b) $\mathbf{S}(\mathcal{P},\ (\curvearrowright l,\ \mathcal{S}_{\mathsf{s}}[\,s\,]),\ m) \rightarrow \mathbf{S}(\mathcal{S}_{\mathsf{s}}\,\mathcal{P},\ (\curvearrowright l,\ s),\ m)$ *if* $l \in \mathsf{labels}\ s$

c) $\mathbf{S}(\mathcal{S}_{\mathsf{s}}\,\mathcal{P},\ (\curvearrowright l,\ s),\ m) \rightarrow \mathbf{S}(\mathcal{P},\ (\curvearrowright l,\ \mathcal{S}_{\mathsf{s}}[\,s\,]),\ m)$ *if* $l \notin \mathsf{labels}\ s$

d) $\mathbf{S}((\mathsf{catch}\ \square)\,\mathcal{P},\ (\uparrow 0,\ s),\ m) \rightarrow \mathbf{S}(\mathcal{P},\ (\nearrow,\ \mathsf{catch}\ s),\ m)$

e) $\mathbf{S}((\mathsf{catch}\ \square)\,\mathcal{P},\ (\uparrow (1+n),\ s),\ m) \rightarrow \mathbf{S}(\mathcal{P},\ (\uparrow n,\ \mathsf{catch}\ s),\ m)$

f) $\mathbf{S}(\mathcal{S}_{\mathsf{s}}\,\mathcal{P},\ (\uparrow n,\ s),\ m) \rightarrow \mathbf{S}(\mathcal{P},\ (\uparrow n,\ \mathcal{S}_{\mathsf{s}}[\,s\,]),\ m)$ *if* $\mathcal{S}_{\mathsf{s}} \neq \mathsf{catch}\ \square$

g) $\mathbf{S}(\mathcal{S}_{\mathsf{s}}\,\mathcal{P},\ (\upuparrows v,\ s),\ m) \rightarrowtriangle \mathbf{S}(\mathcal{P},\ (\upuparrows v,\ \mathcal{S}_{\mathsf{s}}[\,s\,]),\ m)$

7. For block scopes:

a) $\mathbf{S}(\mathcal{P},\ (d,\ \mathsf{local}_{\tau}\ s),\ m) \rightarrowtriangle \mathbf{S}((\mathsf{local}_{o:\tau}\ \Box)\,\mathcal{P},\ (d,\ s),\ \mathsf{alloc}_{\Gamma}\ o\ (\mathsf{new}_{\Gamma}\ \tau)\ \mathsf{false}\ m)$ *if* $(d,\ s)$ in, *for any* $o \notin \mathsf{dom}\ m$

b) $\mathbf{S}((\mathsf{local}_{o:\tau}\ \Box)\,\mathcal{P},\ (d,\ s),\ m) \rightarrowtriangle \mathbf{S}(\mathcal{P},\ (d,\ \mathsf{local}_{\tau}\ s),\ \mathsf{free}\ o\ m)$ *if* $(d,\ s)$ out

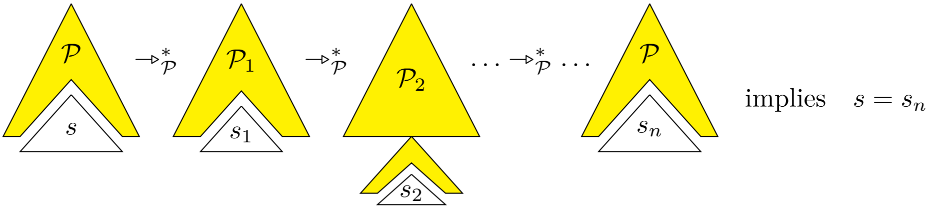
Note that the selection of head redexes in rules 2a, 2b and 2c, and the splitting of $\mathcal{S}_{\mathsf{s}}[\,s\,]$ into $\mathcal{S}_{\mathsf{s}}$ and s in rule 6b is non-deterministic. Arbitrary unused object identifiers are used for local variables in rules 5a and 7a.

Definition 6.5.9. *The* $\mathcal{P}$-restricted small-step reduction $\Gamma, \delta \vdash S_1 \rightarrowtriangle_{\mathcal{P}} S_2$ *is defined as* $\Gamma, \delta \vdash S_1 \rightarrowtriangle S_2$ *provided that* $\mathcal{P}$ *is a suffix of the program context of* S_2*.*

Lemma 6.5.10. *Small-step reduction behaves as a zipper. That is:*

$$\Gamma, \delta \vdash \mathbf{S}(\mathcal{P},\ (d,\ s),\ m) \rightarrowtriangle^{*}_{\mathcal{P}} \mathbf{S}(\mathcal{P},\ (d',\ s'),\ m') \quad \text{implies} \quad s = s'$$

The above lemma shows that the small-step semantics indeed behaves as traversing through a zipper. Its proof is not entirely trivial due to the presence of function calls, which add the statement of the callee to the state. This is shown in the picture:


$\mathcal{P}$
s
$\rightarrowtriangle^{*}_{\mathcal{P}}$
$\mathcal{P}_1$
s_1
$\rightarrowtriangle^{*}_{\mathcal{P}}$
$\mathcal{P}_2$
s_2
$\ldots \rightarrowtriangle^{*}_{\mathcal{P}} \ldots$
$\mathcal{P}$
s_n
implies $s = s_n$

6.6 Type preservation and progress

Well-typedness according to the C type system does not ensure the absence of run-time errors. Even well-typed programs may crash due to *undefined behaviors* such as dereferences of the `NULL` pointer and sequence point violations.

Despite the absence of full type safety, the CH$_2$O core C type system nonetheless enjoys some useful properties. It enjoys type preservation, a weak form of progress, and a weak form of type safety. This weak form of type safety guarantees that each well-typed program can keep on reducing, while possibly eventually reaching a final or undef state. From a programmer's point of view, weak type safety does not provide useful guarantees because a program may still crash due to a run-time error. However, from a formalist's point of view, it ensures that the semantics is well behaved and that we have not forgotten any reductions. In the formalization of LLVM by Zhao *et al.* [ZNMZ12] a similar property is proven.

The type preservation and progress properties deal with arbitrary program states during program execution. We therefore define typing judgments for all components of program states. The main part of a program state $\mathbf{S}(\mathcal{P},\ \phi,\ m)$ is the focus ϕ, for which we define corresponding *focus types*.

Definition 6.6.1. Focus types *are defined as:*

$$\tau_{\mathsf{f}} \in \mathsf{focustype} ::= (\beta,\, \tau^{?}) \mid \tau \mid f$$

Definition 6.6.2. *The judgment* $\Gamma, \Delta \vdash d : (\beta,\, \sigma^{?})$ *describes that* a direction d has type $(\beta,\, \sigma^{?})$. *It is inductively defined as:*

$$\frac{}{\Gamma, \Delta \vdash \nearrow : (\mathsf{false},\, \sigma^{?})} \qquad \frac{\Gamma, \Delta \vdash v : \tau}{\Gamma, \Delta \vdash \upuparrows v : (\beta,\, \tau)} \qquad \frac{d \in \{\searrow, \curvearrowright l, \uparrow n\}}{\Gamma, \Delta \vdash d : (\beta,\, \sigma^{?})}$$

Definition 6.6.3. *The judgment* $\Gamma, \Delta, \vec{\tau} \vdash \phi : \tau_{\mathsf{f}}$ *describes that* a focus ϕ has type τ_{f}. *It is inductively defined as:*

$$\frac{\Gamma, \Delta, \vec{\tau} \vdash s : (\beta,\, \sigma^{?}) \qquad \Gamma, \Delta \vdash d : (\beta,\, \sigma^{?})}{\Gamma, \Delta, \vec{\tau} \vdash (d,\, s) : (\beta,\, \sigma^{?})} \qquad \frac{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e : \tau}{\Gamma, \Delta, \vec{\tau} \vdash e : \tau}$$

$$\frac{\Gamma\, f = (\vec{\sigma},\, \tau) \qquad \Gamma, \Delta \vdash \vec{v} : \vec{\sigma}}{\Gamma, \Delta, \vec{\tau} \vdash \overline{\mathsf{call}}\, f\, \vec{v} : f} \qquad \frac{\Gamma\, f = (\vec{\sigma},\, \tau) \qquad \Gamma, \Delta \vdash v : \tau}{\Gamma, \Delta, \vec{\tau} \vdash \overline{\mathsf{return}}\, f\, v : f}$$

$$\frac{\Gamma, \Delta, \vec{\tau} \vdash e : \tau_{\mathsf{lr}} \qquad \Gamma, \Delta, \vec{\tau} \vdash \mathcal{E} : \tau_{\mathsf{lr}} \rightarrowtail \sigma_{\mathbf{r}}}{\Gamma, \Delta, \vec{\tau} \vdash \overline{\mathsf{undef}}\, (\lightning\, \mathcal{E}\langle e \rangle) : \sigma}$$

$$\frac{\Gamma, \Delta \vdash \Omega \qquad \Gamma, \Delta \vdash v : \tau \qquad \Gamma, \Delta, \vec{\tau} \vdash \mathcal{S}_{\mathsf{e}} : \tau \rightarrowtail (\beta,\, \sigma^{?})}{\Gamma, \Delta, \vec{\tau} \vdash \overline{\mathsf{undef}}\, (\lightning\, \mathcal{S}_{\mathsf{e}}\langle [v]_{\Omega} \rangle) : (\beta,\, \sigma^{?})}$$

The focus type $(\beta,\, \tau^{?})$ is for statement $(s,\, d)$ execution, the focus type is τ for expression execution e, and the focus types f for calling $\overline{\mathsf{call}}\, f\, \vec{v}$ and returning $\overline{\mathsf{return}}\, f\, v$ from functions. The focus for undefined behavior $\overline{\mathsf{undef}}\, \phi_U$ may have either statement or expression type, depending on the source ϕ_U of the undefined behavior.

Definition 6.6.4. *The judgment* $\Gamma, \Delta, \vec{\tau} \vdash \mathcal{P}_{\mathsf{s}} : \tau_{\mathsf{f}} \rightarrowtail \sigma_{\mathsf{f}}$ *describes that* $\mathcal{P}_{\mathsf{s}}$ *is a* valid singular context of type σ_{f} with a hole $\Box$ of type τ_{f}. *It is inductively defined as:*

$$\frac{\Gamma, \Delta, \vec{\tau} \vdash \mathcal{S}_{\mathsf{s}} : (\beta_1,\, \sigma_1^{?}) \rightarrowtail (\beta_2,\, \sigma_2^{?})}{\Gamma, \Delta, \vec{\tau} \vdash \mathcal{S}_{\mathsf{s}} : (\beta_1,\, \sigma_1^{?}) \rightarrowtail (\beta_2,\, \sigma_2^{?})} \qquad \frac{\Delta \vdash o : \tau}{\Gamma, \Delta, \vec{\tau} \vdash \mathsf{local}_{o:\tau}\, \Box : (\beta,\, \sigma^{?}) \rightarrowtail (\beta,\, \sigma^{?})}$$

$$\frac{\Gamma, \Delta, \vec{\tau} \vdash_{\mathbf{r}} e : \tau \qquad \mathsf{locks}\, e = \emptyset \qquad \Gamma, \Delta, \vec{\tau} \vdash \mathcal{S}_{\mathsf{e}} : \tau \rightarrowtail (\beta,\, \sigma^{?})}{\Gamma, \Delta, \vec{\tau} \vdash (\mathcal{S}_{\mathsf{e}}, e) : \tau \rightarrowtail (\beta,\, \sigma^{?})}$$

$$\frac{\Gamma\, f = (\vec{\sigma},\, \tau) \qquad \Gamma, \Delta, \vec{\tau} \vdash \mathcal{E} : \tau_{\mathbf{r}} \rightarrowtail \sigma_{\mathbf{r}}}{\Gamma, \Delta, \vec{\tau} \vdash \mathsf{resume}\, \mathcal{E} : f \rightarrowtail \sigma}$$

$$\frac{\Gamma\, f = (\vec{\sigma},\, \tau) \qquad \Delta \vdash \vec{o} : \vec{\sigma} \qquad (\beta,\, \sigma^{?}) \succ \tau}{\Gamma, \Delta, \vec{\tau} \vdash \mathsf{params}\ f\ \overrightarrow{o\sigma} : (\beta,\, \sigma^{?}) \rightarrowtail f}$$

Definition 6.6.5. *The judgment* $\Gamma, \Delta, \vec{\tau} \vdash \mathcal{P} : \tau_{\mathsf{f}} \rightarrowtail \sigma_{\mathsf{f}}$ *describes that* $\mathcal{P}$ *is a* valid context of type σ_{f} with a hole $\Box$ of type τ_{f}. *It is inductively defined as:*

$$\frac{}{\Gamma, \Delta \vdash \varepsilon : \sigma_{\mathsf{f}} \rightarrowtail \sigma_{\mathsf{f}}} \qquad \frac{\Gamma, \Delta, \mathsf{types}\ (\mathsf{locals}\ \mathcal{P}) \vdash \mathcal{P}_{\mathsf{s}} : \tau_{\mathsf{f}} \rightarrowtail \sigma_{\mathsf{f}} \qquad \Gamma, \Delta \vdash \mathcal{P} : \sigma_{\mathsf{f}} \rightarrowtail \upsilon_{\mathsf{f}}}{\Gamma, \Delta \vdash \mathcal{P}_{\mathsf{s}}\, \mathcal{P} : \tau_{\mathsf{f}} \rightarrowtail \upsilon_{\mathsf{f}}}$$

Fact 6.6.6. *Valid contexts $\mathcal{P}$ have a valid corresponding local stack* $\mathsf{locals}\ \mathcal{P}$*. That means, if* $\Gamma, \Delta \vdash \mathcal{P} : \tau_\mathsf{f} \rightarrowtail \sigma_\mathsf{f}$*, then* $\Delta \vdash \mathsf{locals}\ \mathcal{P}$*.*

The typing judgment for states has the shape $\Gamma \vdash S : g_{\mathtt{main}}$ where $g_{\mathtt{main}} \in \mathsf{funname}$ is the name of the function that started the computation that led to S. For concrete C programs, $g_{\mathtt{main}}$ is typically the `main` function.

Definition 6.6.7. *The judgment* $\Gamma \vdash S : g_{\mathtt{main}}$ *describes that* the state S is valid with respect to an initial function $g_{\mathtt{main}}$. *It is inductively defined as:*

$$\frac{\Gamma, \overline{m}, \mathsf{types}\ (\mathsf{locals}\ \mathcal{P}) \vdash \phi : \tau_\mathsf{f} \qquad \Gamma, \overline{m} \vdash \mathcal{P} : \tau_\mathsf{f} \rightarrowtail g_{\mathtt{main}} \qquad \Gamma \vdash m}{\Gamma \vdash \mathbf{S}(\mathcal{P},\ \phi,\ m) : g_{\mathtt{main}}}$$

Type preservation ensures that if a state S_1 is well-typed, then each subsequent state S_2 after a reduction $S_1 \rightarrowtriangle S_2$ is also well-typed. Type preservation is proven by case analysis on the derivation of the reduction $S_1 \rightarrowtriangle S_2$ and uses the fact that all memory operations preserve typing. The proof is reasonably straightforward with a minor complication.

The compilation is due to the memory environment $\overline{m}$, which appears in the typing judgments $\Gamma, \overline{m}, \mathsf{types}\ (\mathsf{locals}\ \mathcal{P}) \vdash \phi : \tau_\mathsf{f}$ and $\Gamma, \overline{m} \vdash \mathcal{P} : \tau_\mathsf{f} \rightarrowtail g_{\mathtt{main}}$, and is changed into $\overline{m'}$ by a reduction $\mathbf{S}(\mathcal{P},\ \phi,\ m) \rightarrowtriangle \mathbf{S}(\mathcal{P},\ \phi,\ m')$.

Although the contents of a memory m may change arbitrarily during each reduction, the memory environment $\overline{m}$ may not. Only new objects may be allocated and existing objects may be deallocated. Memory indices of deallocated objects cannot be reused for new objects. This evolution is captured by the following relation.

Definition 6.6.8. A memory environment Δ_2 is forward of Δ_1, *notation* $\Delta_1 \Rrightarrow \Delta_2$*, if the following conditions hold:*

1. *If* $\Delta_1 \vdash o : \tau$*, then* $\Delta_2 \vdash o : \tau$*.*
2. *If* $\Delta_1 \vdash o : \tau$ *and* $\Delta_2 \vdash o\ \mathsf{alive}$*, then* $\Delta_1 \vdash o\ \mathsf{alive}$.

Fact 6.6.9. *The relation* $\Rrightarrow$ *is reflexive and transitive.*

Lemma 6.6.10. *All memory operations are forward. That is:*

$$\begin{array}{lll} \overline{m} \Rrightarrow \overline{\mathsf{alloc}_\Gamma\ o\ v\ \beta\ m} & \overline{m} \Rrightarrow \overline{\mathsf{free}\ o\ m} & \overline{m} = \overline{m\langle a := v\rangle_\Gamma} \\ \overline{m} = \overline{\mathsf{force}_\Gamma\ a\ m} & \overline{m} = \overline{\mathsf{lock}_\Gamma\ a\ m} & \overline{m} = \overline{\mathsf{unlock}\ \Omega\ m} \end{array}$$

Fact 6.6.11. *All typing judgments are closed under weakening of typing environments and forwardness of memory typing environments.*

Lemma 6.6.12. *If* $\Gamma \vdash m_1$ *and* $\overline{m}_1 \vdash \rho : \vec{\tau}$ *and* $\Gamma, \overline{m}_1, \vec{\tau} \vdash e_1 : \tau_{\mathsf{lr}}$*, then:*

$$\Gamma, \rho \vdash (e_1, m_1) \rightarrowtriangle_\mathsf{h} (e_2, m_2) \quad \text{implies} \quad \Gamma \vdash m_2,\ \ \Gamma, \overline{m}_2, \vec{\tau} \vdash e_2 : \tau_{\mathsf{lr}} \text{ and } \overline{m}_1 \Rrightarrow \overline{m}_2.$$

Proof. By case analysis on the derivation of $\Gamma, \rho \vdash (e_1, m_1) \rightarrowtriangle_\mathsf{h} (e_2, m_2)$. For each case, we use Lemma 6.6.10 and the fact that the memory operations preserve typing. □

Theorem 6.6.13 (Type preservation). *If* $\Gamma, \overline{\mathsf{mem}\, S_1} \vdash \delta$ *and* $\Gamma \vdash S_1 : g_{\texttt{main}}$*, then:*

$$\Gamma, \delta \vdash S_1 \rightarrowtriangle S_2 \quad \text{implies} \quad \Gamma \vdash S_2 : g_{\texttt{main}}.$$

Proof. By case analysis on the derivation of $\Gamma \vdash S_1 : g_{\texttt{main}}$. We use Lemma 6.6.12 for head reduction. Lemma 6.6.10 and Fact 6.6.11 are used to weaken enclosing parts of the state under changes to the memory. □

Theorem 6.6.14 (Weak progress). *If* $\Gamma, \overline{\mathsf{mem}\, S_1} \vdash \delta$ *and* $\Gamma \vdash S_1 : g_{\texttt{main}}$*, then either:*

1. *Further reduction is possible, that is* $S_1 \rightarrowtriangle S_2$ *for some* S_2*.*
2. *An undefined state is reached, that is* $S_1 = \mathbf{S}(\mathcal{P}, \overline{\mathsf{undef}}\, \phi_U, m)$*.*
3. *A final state is reached, that is* $S_1 = \mathbf{S}(\varepsilon, \overline{\mathsf{return}}\, g\, v, m)$*.*
4. *A label is incorrect, that is* $S_1 = \mathbf{S}(\mathcal{P}, (\curvearrowright l, s), m)$ *with* $l \notin \mathsf{labels}\, s \cup \mathsf{labels}\, \mathcal{P}$*.*
5. *A* ***throw*** *does not have a matching* ***catch****, that is* $S_1 = \mathbf{S}(\mathcal{P}, (\uparrow i, s), m)$ *where* $\mathcal{P}$ *contains fewer surrounding* ***catch****es than* i*.*

Proof. By case analysis on the derivation $\Gamma \vdash S_1 : g_{\texttt{main}}$. □

The judgment $\Gamma \vdash S_1 : g_{\texttt{main}}$ does not ensure the validity of **goto**s and **throw**s. As a result, the progress theorem includes the cases 4 and 5 to account for stuck non-local control. However, the judgment $\Gamma, \Delta \vdash \delta$ for function environments (Definition 6.2.9) does ensure that all **goto**s and **throw**s have a corresponding label and **catch**. So if we start from an initial state, stuck non-local control cannot occur.

Definition 6.6.15. *The* initial state $\mathsf{initial}\ m\ g_{\texttt{main}}\ \vec{v}$ *of a main function* $g_{\texttt{main}}$ *with arguments* $\vec{v}$ *in memory* m *is defined as:*

$$\mathsf{initial}\ m\ g_{\texttt{main}}\ \vec{v} := \mathbf{S}(\epsilon, \overline{\mathsf{call}}\, g_{\texttt{main}}\, \vec{v}, m).$$

Corollary 6.6.16 (Weak type safety). *Suppose* $\Gamma \vdash m$ *and* $\Gamma, \overline{m} \vdash \delta$ *and* $\Gamma\, g_{\texttt{main}} = (\vec{\sigma}, \tau)$ *and* $\Gamma, \overline{m} \vdash \vec{v} : \vec{\sigma}$*. Now if:*

$$\Gamma, \delta \vdash \mathsf{initial}\ m\ g_{\texttt{main}}\ \vec{v} \rightarrowtriangle^* S$$

then we have either:

1. *Further reduction is possible, that is* $S \rightarrowtriangle S'$ *for some* S'*.*
2. *An undefined state is reached, that is* $S = \mathbf{S}(\mathcal{P}, \overline{\mathsf{undef}}\, \phi_U, m)$*.*
3. *A final state is reached, that is* $S = \mathbf{S}(\varepsilon, \overline{\mathsf{return}}\, g\, v, m)$*.*

Proof. Using Theorems 6.6.14 and 6.6.13, and the fact that $\rightarrowtriangle$ preserves validity of **goto**s and **throw**s. □

All typing judgments in this thesis have a corresponding type inference function. These type inference functions have been formally defined in the Coq development but are not used explicitly in this thesis. After all, the translation from CH$_2$O abstract C into CH$_2$O core C (Section 7.2) ensures that each CH$_2$O core C program that we obtain from C sources is well-typed by construction (Theorem 7.2.5 on page 137). Since these type inference functions either yield a unique type in case the input is typeable or fail otherwise, they give rise to uniqueness of typing.

Lemma 6.6.17. *All typing judgments satisfy uniqueness of typing and have a corresponding type inference function.*

Proof. Since all typing judgments are defined in a syntax directed way, type inference functions can be defined in the obvious way. Uniqueness of typing is a corollary. □

6.7 Invariance under memory refinements

We show that the operational semantics is invariant under memory refinements. This is a good sanity check to ensure that the operational semantics does not expose internal properties of the memory representation. Besides, it opens the door to reasoning about other kind of program transformations in future work.

To prove invariance of the operational semantics under memory refinements, we lift memory refinements $m_1 \sqsubseteq_\Gamma^f m_2$ (Section 5.8) to state refinements $S_1 \sqsubseteq_\Gamma^f S_2 : g_{\mathtt{main}}$ where $g_{\mathtt{main}}$ denotes the main function. Refinement invariance is stated in terms of a backward simulation because of non-determinism.

Theorem 6.7.1 (Refinement invariance)**.** *If* $\Gamma, \delta \vdash S_1 \rightarrowtriangle S_2$ *and* $S_1' \sqsubseteq_\Gamma^f S_1 : g_{\mathtt{main}}$ *and* S_1 *is not an* $\overline{\mathsf{undef}}$ *state, then there exists an* $f' \supseteq f$ *and* S_2' *with:*

$$\begin{tikzcd}
S_1' \arrow[r, dashed, -Triangle] \arrow[d, no head, "\sqsubseteq_\Gamma^f"'] & S_2' \arrow[d, dashed, no head, "\sqsubseteq_\Gamma^{f'}"'] \\
S_1 \arrow[r, -Triangle] & S_2
\end{tikzcd}$$

Proof. By case analysis on the derivation of $S_1 \rightarrowtriangle S_2$. Steps involving memory operations require invariance properties of the memory model. □

When considering the correctness proof of a compiler, the reductions on the bottom of the diagram correspond to those of the target program, and the reductions on the top correspond to those of the source program. The backward simulation has the desired consequence that if the target program can make a step, then either:

- The source program is already in an $\overline{\mathsf{undef}}$ state due to prior undefined behavior. Since undefined behavior subsumes all behaviors, the target program may do anything in this case.
- The source program can make a corresponding step.

The relation $\sqsubseteq_\Gamma^f$ goes from the top to the bottom because a compiler is allowed to make the memory more defined. For example, it is allowed to assign a concrete value to an object that has an indeterminate value in the source.

6.8 Executable semantics

The operational semantics of CH$_2$O core C is defined as an inductively defined reduction relation $\Gamma, \delta \vdash \_ \rightarrowtriangle \_ : \mathsf{state} \to \mathsf{state} \to \mathsf{Prop}$. Since this relation is not executable, we define a function $\mathsf{exec}_{\Gamma,\delta} : \mathsf{state} \to \mathcal{P}_{\mathsf{fin}}(\mathsf{state})$ that *computes* a corresponding finite set of subsequent states up to renaming of object identifiers.

Given that we have already defined basic operations, such as accessing the memory and performing an operator, as Coq functions that are effectively executable, creating an executable version of our operational semantics is largely straightforward. However, non-determinism makes the situation more complicated.

- Execution of expressions is non-deterministic. For example, the two assignments in `(*p = 1) + (*q = 2)` may be executed in any order. This is described by the following rule (rule 2a of Definition 6.5.8):

  $$\mathbf{S}(\mathcal{P},\ \mathcal{E}[\,e_1\,],\ m_1) \twoheadrightarrow \mathbf{S}(\mathcal{P},\ \mathcal{E}[\,e_2\,],\ m_2)$$

  Here, an expression is decomposed non-deterministically in an evaluation context $\mathcal{E}$ and a subexpression e_1.
- The choice of picking an unused object identifier for newly allocated memory is non-deterministic. For example, this occurs in the rule for entering a block scope with a local variable (rule 7a of Definition 6.5.8):

  $$\mathbf{S}(\mathcal{P},\ (\searrow,\ \mathsf{local}_\tau\ s),\ m) \twoheadrightarrow \mathbf{S}((\mathsf{local}_{o:\tau}\ \square)\,\mathcal{P},\ (\searrow,\ s),\ \mathsf{alloc}_\Gamma\ o\ (\mathsf{new}_\Gamma\ \tau)\ \mathsf{false}\ m)$$

  Here, any unused object identifier $o \notin \mathsf{dom}\ m$ may be chosen.

The first source of non-determinism is finitary as expressions can only be decomposed in a finite number of ways. However, there is an infinite choice of picking fresh object identifiers.

In order to deal with the first source of non-determinism, we define a function $\mathsf{redexes} : \mathsf{expr} \to \mathcal{P}_{\mathsf{fin}}(\mathsf{ectx} \times \mathsf{expr})$ that decomposes an expression into a finite set of all possible combinations of evaluation contexts and redexes. This function is defined as expected, so we just state its defining property.

Lemma 6.8.1 (Soundness and completeness of redex splitting)**.** *We have:*

$$(\mathcal{E},\ e) \in \mathsf{redexes}\ e' \quad \text{iff} \quad e' = \mathcal{E}[\,e\,] \text{ and } e \text{ is a redex.}$$

The notion of a redex is defined in Definition 6.4.6.

The second source of non-determinism is more difficult to deal with. Since there is an infinite choice of fresh object identifiers, we cannot enumerate them all. Instead, we will just pick one and prove that the semantics is invariant under renaming of object identifiers. The corresponding clause of the executable semantics thus becomes:

$$\begin{aligned}\mathsf{exec}_{\Gamma,\delta}\ (\mathbf{S}(\mathcal{P}, (\searrow, \mathsf{local}_\tau\ s), m)) := \mathbf{let}\ o := \mathsf{fresh}\ m\ \mathbf{in}& \\ \{\mathbf{S}((\mathsf{local}_{\tau:o}\ \square)\,\mathcal{P},\ (\searrow, s),\ \mathsf{alloc}_\Gamma\ o\ (\mathsf{new}_\Gamma\ \tau)\ \mathsf{false}\ m)\}&\end{aligned}$$

By using a canonical object identifier for newly allocated memory, we have removed the second source of non-determinism entirely.

Since the operational semantics is defined in an almost syntax directed fashion, the exact definition of the executable semantics is very close to the operational semantics. In the Coq development, we have implemented it in monadic style [Mog89, Wad90] using the *finite set monad*.

Reducing the amount of non-determinism by using a canonical object identifier for each newly allocated object does not affect the soundness theorem.

Theorem 6.8.2 (Soundness of the executable semantics)**.** *If $S_2 \in \mathsf{exec}_{\Gamma,\delta}\ S_1$, then $\Gamma, \delta \vdash S_1 \rightarrowtriangle S_2$.*

Proof. By case analysis on S_1 using Lemma 6.8.1. □

The converse of this theorem is not true. Given a reduction $\Gamma, \delta \vdash S_1 \rightarrowtriangle S_2$ in the operational semantics, we do not necessarily have $S_2 \in \mathsf{exec}_{\Gamma,\delta}\ S_1$ in the executable semantics as well. That is because a different object identifier than the canonical one could have been used. To prove completeness, we show that the semantics is invariant under this choice.

Definition 6.8.3. *A state S_1 is an f-*permutation *of state S_2, notation $S_1 \sim_f S_2$, if S_2 is obtained by renaming the object identifiers in S_1 with respect to $f : \mathsf{index} \to \mathsf{option\ index}$.*

Fact 6.8.4. *We have the following properties:*

1. Identity: *$S \sim_f S$ for any f with $f\,x = x$ for each $x \in \mathsf{dom}\ S$.*

2. Composition: *If $S_1 \sim_f S_2$ and $S_2 \sim_{f'} S_3$, then $S_1 \sim_{f' \circ f} S_3$.*

3. Symmetry: *If $S_1 \sim_f S_2$, then $S_2 \sim_{f^{-1}} S_1$.*

4. Weakening: *If $S_1 \sim_f S_2$ and $f' \supseteq f$, then $S_1 \sim_{f'} S_2$.*

We first prove that the operational semantics is invariant under permutations (Lemma 6.8.5). Completeness is then obtained using this result combined with the fact that a single step in the executable semantics yields a permutation (Lemma 6.8.6).

Lemma 6.8.5. *If $\Gamma, \delta \vdash S_1 \rightarrowtriangle S_2$ and $S_1' \sim_f S_1$, then there exists an $f' \supseteq f$ and S_2' such that:*

$$\begin{tikzcd}
S_1' \arrow[r, dashed, -Triangle] \arrow[d, dash, "\sim_f"] & S_2' \arrow[d, dash, dashed, "\sim_{f'}"] \\
S_1 \arrow[r, -Triangle] & S_2
\end{tikzcd}$$

Proof. By case analysis on the derivation of $S_1 \rightarrowtriangle S_2$. Steps involving memory operations require commuting properties with respect to permutations. □

In our Coq development, permutations and memory refinements (Section 5.8) are generalized to a single notion in order to avoid duplication of code. The previous theorem therefore becomes an instance of Theorem 6.7.1. In this thesis we present both notions separately for brevity's sake.

Lemma 6.8.6. *If $\Gamma, \delta \vdash S_1 \rightarrowtriangle S_2$, then there exists an f and S_2' such that:*

$$\begin{tikzcd}
 & S_2' \arrow[d, dash, dashed, "\sim_f"] \\
S_1 \arrow[ur, dashed, -Triangle, "\mathsf{exec}"] \arrow[r, -Triangle] & S_2
\end{tikzcd}$$

Here, $S_1 \overset{\mathsf{exec}}{\rightarrowtriangle} S_2'$ denotes $S_2' \in \mathsf{exec}_{\Gamma,\delta}\ S_1$.

Proof. By case analysis on the derivation of $S_1 \rightarrowtriangle S_2$. Lemma 6.8.1 is used for expression steps, and steps involving allocation require properties about preservation of alloc_Γ with respect to permutations. □

Theorem 6.8.7 (Completeness of the executable semantics)**.** *If* $\Gamma, \delta \vdash S_1 \rightarrowtriangle^* S_2$*, then there exists an* f *and* S_2' *such that:*

$$\begin{tikzcd}
 & S_2' \arrow[d, dash, dashed, "\sim_f"] \\
S_1 \arrow[ur, dashed, "\mathsf{exec}", "*" near end] \arrow[r, "*"'] & S_2
\end{tikzcd}$$

Proof. By induction from right to left on the derivation of $S_1 \rightarrowtriangle^* S_2$. The inductive case is depicted in the following diagram:

$$\begin{tikzcd}
 & & S_3'' \arrow[d, dash, dashed, "\sim_{f_2}"] \arrow[dd, dash, dashed, bend left=60, "\sim_{f_1 \circ f_2}"] \\
 & S_2' \arrow[ur, dashed, "\mathsf{exec}"] \arrow[r, dashed, "\text{Lemma 6.8.6}"] \arrow[d, dash, dashed, "\sim_f"] & S_3' \arrow[d, dash, dashed, "\sim_{f_1}"] \arrow[r, phantom, "\text{Lemma 6.8.4}"] & {} \\
S_1 \arrow[ur, dashed, "\mathsf{exec}", "*" near end] \arrow[r, "\text{IH}", "*"' near end] & S_2 \arrow[r] \arrow[ur, phantom, "\text{Lemma 6.8.5}"] & S_3
\end{tikzcd}$$

□

Related work. There have been many efforts to define an executable version of the semantics of a real-life programming language in a proof assistant.

Campbell has defined an executable semantics for CompCert C [Cam12]. He has proven soundness and completeness with respect to a deterministic left-to-right operational semantics of CompCert C. Campbell's executable semantics has been reimplemented by Leroy and is currently part of the official CompCert distribution. Leroy's reimplementation handles non-determinism and is thereby able to compute all possible behaviors of a given program. Our completeness proof is more complicated than CompCert's because our operational semantics involves infinitary branching whereas non-determinism in CompCert C is at most finitarily branching.

Furthermore, Zhao *et al.* have created an executable semantics for a deterministic fragment of LLVM [ZNMZ12], Bodin *et al.* have created an executable semantics for Javascript [BCF$^+$14], and Lochbihler and Bulwahn have created an executable semantics for Java [LB11]. Lochbihler and Bulwahn use code extraction to generate an executable semantics from an inductive definition. We have written our executable semantics by hand due to excessive non-determinism in C.

There has also been related work on tool support for writing down programming languages definitions in a domain specific language that can be automatically translated into an executable semantics, a formal definition in some proof assistant or a LaTeX document. See for example the OTT [SNO$^+$07] and Lem [OBNS11] tools. Since we wish to use advanced features of Coq, we have not used such tools.

6.9 Evaluation of pure expressions

Expressions of CH$_2$O core C contain constructs that involve side-effects. Side-effects include assignments $e_1\ \alpha\ e_2$, function calls $e(\vec{e})$, and allocation $\mathsf{alloc}_\tau\ e$ and deallocation $\mathsf{free}\ e$ of dynamically obtained memory. In this section we consider the subset of expressions that is side-effect free. These expressions are called *pure*.

Pure expressions have three essential properties: they do not modify the memory, they have a unique result, and their executions always terminate. These properties allow us to define an evaluator $\llbracket\,\_\,\rrbracket_{\Gamma,\rho,m} : \mathsf{expr} \to \mathsf{option}\ \mathsf{lrval}$ that fails in case of undefined behavior, and yields the resulting address or value otherwise.

The pure expression evaluator will be used for two rather different purposes in this thesis:

- Is will be used for constant expression evaluation during the translation from CH$_2$O abstract C into CH$_2$O core C (Section 7.2). Constant expressions appear for example in initializers of global variables and array sizes [ISO12, 6.6].
- It will be used to relate expressions to their results in our separation logic for CH$_2$O core C (Chapter 8). The assertion $e \Downarrow \nu$ expresses that an expression e evaluates to ν (Definition 8.2.7 on page 151).

In this section we will prove soundness and completeness of the expression evaluator with respect to the operational semantics. Soundness is essential for the soundness proof of separation logic. Soundness and completeness are desirable sanity properties of constant expressions evaluation.

The evaluator is defined by structural recursion over the structure of expressions. It yields $\perp$ if the given expression is not pure or if it has undefined behavior.

Definition 6.9.1. Evaluation of constant expressions $\llbracket\,\_\,\rrbracket_{\Gamma,\rho,m} : \mathsf{expr} \to \mathsf{option}\ \mathsf{lrval}$ *is defined as:*

$$\begin{array}{rll}
\llbracket x_i \rrbracket & := \mathsf{top}_\tau\, o & \text{if } \rho(i) = (o,\, \tau) \\
\llbracket [\nu]_\emptyset \rrbracket & := \nu & \\
\llbracket *e \rrbracket & := p & \text{if } \llbracket e \rrbracket = \mathsf{ptr}\, p \text{ and } \overline{m} \vdash p\ \mathsf{alive} \\
\llbracket \& e \rrbracket & := \mathsf{ptr}\, p & \text{if } \llbracket e \rrbracket = p \\
\llbracket \mathsf{load}\ e \rrbracket & := m\langle a\rangle_\Gamma & \text{if } \llbracket e \rrbracket = a \text{ and } \mathsf{force}_\Gamma\ a\ m = m \\
\llbracket e\ ._{\mathbf{l}}\ r \rrbracket & := a\langle r\rangle_\Gamma & \text{if } \llbracket e \rrbracket = a \text{ and } \Gamma \vdash a\ \mathsf{strict} \\
\llbracket e\ ._{\mathbf{r}}\ r \rrbracket & := v[r]_\Gamma & \text{if } \llbracket e \rrbracket = v \\
\llbracket e_1[\vec{r} := e_2] \rrbracket & := v_1[\vec{r} := v_2]_\Gamma & \text{if } \llbracket e_1 \rrbracket = v_1,\ \llbracket e_2 \rrbracket = v_2 \text{ and } v_1[\vec{r}]_\Gamma \neq \perp \\
\llbracket \circledcirc_u\, e \rrbracket & := \circledcirc_u\ v & \text{if } \llbracket e \rrbracket = v \text{ and } m \vdash (\circledcirc_u\, v)\ \mathsf{defined} \\
\llbracket e_1 \circledcirc e_2 \rrbracket & := v_1 \circledcirc v_2 & \text{if } \llbracket e_1 \rrbracket = v_1,\ \llbracket e_2 \rrbracket = v_2 \text{ and } \Gamma, m \vdash (v_1 \circledcirc v_2)\ \mathsf{defined} \\
\llbracket (e_1, e_2) \rrbracket & := \llbracket e_2 \rrbracket & \text{if } \llbracket e_1 \rrbracket \neq \perp
\end{array}$$

$$\llbracket e_1\ ?\ e_2 : e_3 \rrbracket := \begin{cases} \llbracket e_2 \rrbracket & \text{if } \llbracket e_1 \rrbracket = v_{\mathsf{b}},\ m \vdash (\mathsf{zero}\ v_{\mathsf{b}})\ \mathsf{defined} \text{ and } \neg\mathsf{zero}\ v_{\mathsf{b}} \\ \llbracket e_3 \rrbracket & \text{if } \llbracket e_1 \rrbracket = v_{\mathsf{b}},\ m \vdash (\mathsf{zero}\ v_{\mathsf{b}})\ \mathsf{defined} \text{ and } \mathsf{zero}\ v_{\mathsf{b}} \end{cases}$$

Evaluation of a $\mathsf{load}\ e$ expression has the side-condition $\mathsf{force}_\Gamma\ a\ m = m$ that arises from the fact that a load is not necessarily pure. Although a load does not change the memory contents, it may still affect the effective types and thereby change the variants of unions in memory (see Definition 5.4.13 on page 78). The side-condition $\mathsf{force}_\Gamma\ a\ m = m$ thus ensures that the given load is pure.

Soundness (Theorem 6.9.5) and completeness (Theorem 6.9.7) with respect to the operational semantics correspond to the following statement:

$$\llbracket e \rrbracket_{\Gamma,\mathsf{locals}\ \mathcal{P},m} = \nu \quad \text{iff} \quad \Gamma,\delta \vdash \mathbf{S}(\mathcal{P},\ e,\ m) \mathrel{-\!\!\triangleright}^*_{\mathcal{P}} \mathbf{S}(\mathcal{P},\ [\nu]_\emptyset,\ m).$$

Soundness (left to right) holds unconditionally. If an expression e is not covered by the evaluator, for example if it contains an assignment or a function call, we have $\llbracket e \rrbracket_{\Gamma,\mathsf{locals}\ \mathcal{P},m} = \bot$, and so the statement holds trivially.

Completeness (right to left) only holds if the given expression is actually *pure*. For example, expressions that contain an assignment may have a corresponding reduction in the operational semantics, but are not covered by the evaluator. To that end, we define a predicate that carves out the set of pure expressions.

Definition 6.9.2. *The judgment* e pure *denotes that* an expression e is considered pure*. It is inductively defined as:*

$$\frac{}{x_i\ \mathsf{pure}} \qquad \frac{}{[\nu]_\emptyset\ \mathsf{pure}} \qquad \frac{e\ \mathsf{pure}}{(*e)\ \mathsf{pure}} \qquad \frac{e\ \mathsf{pure}}{(\&e)\ \mathsf{pure}} \qquad \frac{e\ \mathsf{pure}}{(e \mathrel{.}_{\mathbf{r}} r)\ \mathsf{pure}}$$

$$\frac{e\ \mathsf{pure}}{(e \mathrel{.}_{\mathbf{l}} r)\ \mathsf{pure}} \qquad \frac{e_1\ \mathsf{pure} \qquad e_2\ \mathsf{pure}}{e_1[\vec{r} := e_2]\ \mathsf{pure}} \qquad \frac{e\ \mathsf{pure}}{\circledcirc_u\, e\ \mathsf{pure}} \qquad \frac{e_1\ \mathsf{pure} \qquad e_2\ \mathsf{pure}}{(e_1 \circledcirc e_2)\ \mathsf{pure}}$$

$$\frac{e\ \mathsf{pure}}{(\tau)e\ \mathsf{pure}} \qquad \frac{e_1\ \mathsf{pure} \qquad e_2\ \mathsf{pure}}{(e_1, e_2)\ \mathsf{pure}} \qquad \frac{e_1\ \mathsf{pure} \qquad e_2\ \mathsf{pure} \qquad e_3\ \mathsf{pure}}{(e_1\ ?\ e_2 : e_3)\ \mathsf{pure}}$$

Fact 6.9.3. *For each expression e, we have either:*

1. *It is a value, that is $e = [\nu]_\Omega$.*
2. *It contains a redex, that is $e = \mathcal{E}[\, e' \,]$ where e' a redex.*

Lemma 6.9.4. *Given a redex e_1 with $\llbracket \mathcal{E}[\, e_1 \,] \rrbracket_{\Gamma,\rho,m} = \nu$, then there exists an e_2 with:*

$$\Gamma,\rho \vdash (e_1, m) \mathrel{-\!\!\triangleright}_{\mathsf{h}} (e_2, m) \quad \text{and} \quad \llbracket \mathcal{E}[\, e_2 \,] \rrbracket_{\Gamma,\rho,m} = \nu$$

Theorem 6.9.5 (Soundness of expression evaluation)**.** *For each expression e we have:*

$$\llbracket e \rrbracket_{\Gamma,\mathsf{locals}\ \mathcal{P},m} = \nu \quad \text{implies} \quad \Gamma,\delta \vdash \mathbf{S}(\mathcal{P},\ e,\ m) \mathrel{-\!\!\triangleright}^*_{\mathcal{P}} \mathbf{S}(\mathcal{P},\ [\nu]_\emptyset,\ m).$$

Proof. We use well-founded induction on the size of e. By Fact 6.9.3 we know that e is either a value or contains a redex. In the first case, we are done. In the second case, we use Lemma 6.9.4 to perform a reduction in the operational semantics and proceed by induction. □

Lemma 6.9.6. *If $\Gamma,\rho \vdash (e_1, m_1) \mathrel{-\!\!\triangleright}_{\mathsf{h}} (e_2, m_2)$ and e_1 pure, then:*

$$m_1 = m_2 \quad \text{and} \quad \llbracket e_1 \rrbracket_{\Gamma,\rho,m_1} = \llbracket e_2 \rrbracket_{\Gamma,\rho,m_2} \quad \text{and} \quad e_2\ \mathsf{pure}.$$

Theorem 6.9.7 (Completeness of pure expression evaluation)**.** *For each expression e with e* pure *we have:*

$$\Gamma, \delta \vdash \mathbf{S}(\mathcal{P},\, e,\, m) \rightarrow^{*}_{\mathcal{P}} \mathbf{S}(\mathcal{P},\, [\nu]_{\emptyset},\, m) \quad \text{implies} \quad [\![\, e \,]\!]_{\Gamma, \mathsf{locals}\ \mathcal{P}, m} = \nu.$$

Proof. We use induction on the derivation $\Gamma, \delta \vdash \mathbf{S}(\mathcal{P},\, e,\, m) \rightarrow^{*}_{\mathcal{P}} \mathbf{S}(\mathcal{P},\, [\nu]_{\emptyset},\, m)$. We use Lemma 6.9.6 in each step. □

CHAPTER 7

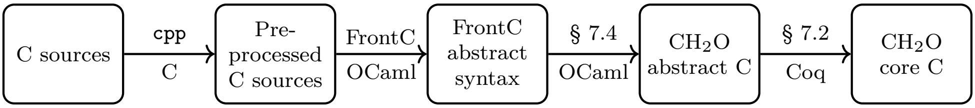
CH_2O abstract C

This chapter describes the language CH_2O abstract C whose syntax closely resembles the structure of C source files. This language bridges the gap between our simplified C language CH_2O core C and actual C abstract syntax trees. CH_2O abstract C uses named variables instead of De Bruijn indices, supports global and static variables, initializers, enum types, typedefs, and C-like looping statements. The semantics of CH_2O abstract C is specified by translation into CH_2O core C.

This chapter furthermore presents an ‘interpreter’ based on our executable semantics that allows one to test the CH_2O semantics. This is not an ordinary interpreter that executes a given program according to *one* interpretation of the C11 standard and has arbitrary behavior when undefined behavior occurs. Instead, our interpreter calculates *all* behaviors of a given program. When the program has undefined behavior, our interpreter will explicitly state this undefinedness.

The CH_2O interpreter is also different from compilers and interpreters that insert tests for undefined behaviors as a protection. Those compilers and interpreters tend to follow one specific execution order. Instead, the CH_2O interpreter is not primarily meant to be a debugging tool, but instead is considered an *exploration* tool, intended to explore the implications of the C11 standard.

The CH_2O interpreter consists of four phases to transform C source code into a corresponding CH_2O core C program:



The first phase uses the GNU C preprocessor (called `cpp` in the diagram) to perform macro expansion and inclusion of header files. The second phase then uses the FrontC parser to transform the preprocessed C sources into an OCaml representation of an abstract syntax tree.

The third phase is a thin layer of OCaml glue that transforms the abstract syntax tree into a CH_2O abstract C program. In turn, the fourth phase transforms the CH_2O abstract C program into a CH_2O core C program. This CH_2O core C program is run using the executable semantics.

From a semanticist's point of view, the interesting work is performed by the fourth phase, which is written fully in Coq. Program extraction from Coq into OCaml [Let04] is used to combine it with the other phases into a standalone tool.

Since the fourth phase is written in Coq, it is guaranteed to be well-defined. That means, it always terminates and failure is tightly controlled by an error monad. Moreover, we have used Coq to prove that whenever it succeeds to produce a $\mathrm{CH_2O}$ core C program, this program is well-typed.

Before we will present $\mathrm{CH_2O}$ abstract C and its translation into $\mathrm{CH_2O}$ core C we consider a small example:

```
struct S { char c; short h; } s = { .h = CHAR_BIT };
static int i = sizeof(s);
int main() {
  int j = 2L, *k = 0;
  return j = (i = j++);
}
```

After parsing and the translation of the AST into $\mathrm{CH_2O}$ abstract C, we obtain a sequence of declarations Θ. These declarations correspond closely to the C sources and still contain implementation-defined constructs such as the **sizeof** expression:

$$\begin{aligned}
\Theta := {} & (\mathtt{S}, \mathsf{struct}\ [\,\mathsf{char}\ \mathtt{c},\ \ \mathsf{short}\ \mathtt{h}\,]),\\
& (\mathtt{s}, \mathsf{global}\ \{.\mathtt{h} := \mathsf{char\,bits}\}\ :\ \ \mathsf{struct}\ \mathtt{S}),\\
& (\mathtt{i}, \mathsf{global}\ (\mathsf{sizeof}\ \ (\mathsf{typeof}\ \mathtt{s}))\ :\mathsf{static\ int}),\\
& (\mathtt{main}, \mathsf{fun}\ (\\
& \quad \mathsf{int}\ \mathtt{j} := \mathsf{const}_{\mathsf{long}}\ 2\ ;\ \ \mathsf{int}\ \mathtt{k} := \mathsf{const}_{\mathsf{int}}\ 0\ ;\\
& \quad \mathsf{return}\ \mathtt{j} := (\mathtt{i} := (\mathtt{j} \mathrel{+}:= \mathsf{const}_{\mathsf{int}}\ 1))\\
&)\ :\ \ \varepsilon \to \mathsf{int})
\end{aligned}$$

The translation from $\mathrm{CH_2O}$ abstract C transforms the declarations Θ into a typing environment Γ, an initial memory m and a $\mathrm{CH_2O}$ core C program δ:

$$\begin{aligned}
\Gamma &:= \mathtt{S} : [\,\mathsf{signed\ char},\ \ \mathsf{signed\ short}\,],\ \mathtt{main} : (\epsilon,\ \mathsf{signed\ int})\\
m &:= 0 \mapsto \mathsf{ofval}_{\Gamma}\ (\Diamond(0,\ 1)^{32})\ (\mathsf{struct}_{\mathtt{S}}\ [\,\mathsf{int}_{\mathsf{signed\ char}}\ 0, \mathsf{int}_{\mathsf{signed\ short}}\ 8\,]),\\
&\qquad 1 \mapsto \mathsf{ofval}_{\Gamma}\ (\Diamond(0,\ 1)^{32})\ (\mathsf{int}_{\mathsf{signed\ int}}\ 4)\\
\delta &:= \mathtt{main} \mapsto \left(\begin{array}{l}
\mathsf{local}_{\mathsf{signed\ int}}\ (x_0 := \mathsf{int}_{\mathsf{signed\ long}}\ 2\ ;\\
\quad \mathsf{local}_{\mathsf{signed\ int*}}\ (x_0 := \mathsf{ptr}\ (\mathsf{NULL}\ (\mathsf{signed\ int}))\ ;\\
\quad\quad \mathsf{return}\ (x_1 := (\mathsf{top}_{\mathsf{signed\ int}}\ 1 := (x_1 \mathrel{+}:= \mathsf{int}_{\mathsf{signed\ int}}\ 1)))\\
\quad)\\
)
\end{array}\right)
\end{aligned}$$

As shown, global variables are translated into memory indices, named variables are translated into De Bruijn indices, and various type annotations have been added. Moreover, implementation-defined constructs such as the signedness of char and the value of the sizeof have been concretized. Type soundness of the translation guarantees that we have $\vdash \Gamma$ and $\Gamma \vdash m$ and $\Gamma, \overline{m} \vdash \delta$.

7.1 Syntax

This section defines the language CH$_2$O abstract C, which is called that way because its syntax is very close to C abstract syntax trees. The semantics will be defined in Section 7.2 through a translation into CH$_2$O core C.

Definition 7.1.1. CH$_2$O abstract C integer types *are inductively defined as:*

$$\begin{aligned}
k \in \mathsf{cintrank} &::= \mathsf{char} \mid \mathsf{short} \mid \mathsf{int} \mid \mathsf{long} \mid \mathsf{long\ long} \mid \mathsf{ptr} \\
si \in \mathsf{signedness} &::= \mathsf{signed} \mid \mathsf{unsigned} \\
\tau_\mathsf{i} \in \mathsf{cinttype} &::= si^?\ k
\end{aligned}$$

These integer types differ in two ways from the ones of CH$_2$O core C. Signedness is optional and ranks are syntactic instead of abstract entities of an implementation environment. Integer types are signed by default [ISO12, 6.2.5p4], with the exception of char, whose signedness is implementation-defined [ISO12, 6.2.5p15].

Definition 7.1.2. CH$_2$O abstract C types, expressions, reference segments and initializers *are mutually inductively defined as:*

$$\begin{aligned}
\tau \in \mathsf{ctype} &::= \mathsf{void} \mid \mathsf{def}\, x \mid \tau_\mathsf{i} \mid \tau* \mid \overrightarrow{\tau\, x^?} \to \tau \mid \tau[e] \\
&\quad\ \mid \mathsf{struct}\, x \mid \mathsf{union}\, x \mid \mathsf{enum}\, x \mid \mathsf{typeof}\, e \\
e \in \mathsf{cexpr} &::= x \mid \mathsf{const}_{\tau_\mathsf{i}}\ z \mid \mathsf{string}\ \vec{z} && \text{(variables, integer and string constants)} \\
&\quad\ \mid \mathsf{sizeof}\ \tau \mid \mathsf{alignof}\ \tau \mid \mathsf{offsetof}\ \tau\ x \\
&\quad\ \mid \tau\, \mathsf{min} \mid \tau\, \mathsf{max} \mid \tau\, \mathsf{bits} && \text{(implementation-defined constants)} \\
&\quad\ \mid \&e \mid *e && \text{(address of and dereference operator)} \\
&\quad\ \mid e\ .\ x && \text{(indexing of structs and unions)} \\
&\quad\ \mid e_1\ \alpha\ e_2 && \text{(assignments)} \\
&\quad\ \mid e(\vec{e}) \mid && \text{(function calls)} \\
&\quad\ \mid \mathsf{alloc}_\tau\ e \mid \mathsf{free}\ e && \text{(allocation and deallocation)} \\
&\quad\ \mid \circledcirc_u\ e \mid e_1 \circledcirc e_2 \mid (\tau)I && \text{(unary, binary and cast operators)} \\
&\quad\ \mid e_1\ \texttt{\&\&}\ e_2 \mid e_1\ \texttt{||}\ e_2 \mid (e_1, e_2) \mid e_1\ ?\ e_2 : e_3 && \text{(sequenced operators)} \\
r \in \mathsf{crefseg} &::= [e] \mid .x \\
I \in \mathsf{cinit} &::= e \mid \{\overrightarrow{\vec{r} := I}\}
\end{aligned}$$

Types and expressions of CH$_2$O abstract C are mutually inductive. Most notably, the size of an array type $\tau[e]$ may be specified by an expression rather than an integer literal. These expressions are *constant expressions* [ISO12, 6.6] and are replaced by their actual values during the translation into CH$_2$O core C.

Typedefs are abbreviations of types that can be declared throughout the program. The following code introduces `string` as an abbreviation of `char*`:

```
typedef char *string;
int length(string p);
```

The type construct $\mathsf{def}\ x$ is used to refer to a typedef named x. The CH$_2$O abstract C type of the function `length` is thus $[\,(\mathsf{def}\ \texttt{string})\ \texttt{p}\,] \to \mathsf{int}$.

Enum types are lists of integer constants. For example:

```
enum color {
  red, green,       // red has value 0, green has value 1
  blue = green + 3, // blue has value 4
  yellow            // yellow has value 5
} x = red;
```

The enum declaration introduces integer constants `red`, `green`, `blue` and `yellow`. The first constant has value 0, the second 1, and so on. Constants can be given an explicit value after which the progression continues.

Expressions contain constants that denote implementation-defined integer values. These constants are as follows:

CH$_2$O	C11	Value it denotes
$\mathsf{sizeof}\ \tau$	`sizeof`(τ)	size of a type τ
$\mathsf{alignof}\ \tau$	`_Alignof`(τ)	alignment of a type τ
$\mathsf{offsetof}\ \tau\ x$	`offsetof`$(\tau,\ x)$	offset of a field x in a struct or union τ
$\tau\ \mathsf{min}$	`SHRT_MIN`, `INT_MIN`, *etc.*	minimal value of an integer type τ
$\tau\ \mathsf{max}$	`INT_MAX`, `UINT_MAX`, *etc.*	maximal value of an integer type τ
$\tau\ \mathsf{bits}$	`CHAR_BIT`	number of bits of an integer type τ

We do not use the preprocessor to substitute these constructs for concrete values, but instead let the translation into CH$_2$O core C handle this. This translation fetches the concrete value from the used implementation environment.

In CH$_2$O abstract C, as well as in actual C, the integer literal $\mathsf{const}_{\tau_i}\ 0$ is overloaded to denote both the `NULL` pointer and the integer constant 0 [ISO12, 6.3.2.3p3]. A string literal $\mathsf{string}\ \vec{z}$ designates a character arrays with read only permission.

Definition 7.1.3. CH$_2$O abstract C statements *are inductively defined as:*

$$
\begin{array}{rcll}
sto \in \mathsf{cstorage} & ::= & \mathsf{static} \mid \mathsf{extern} \mid \mathsf{auto} & \\
s \in \mathsf{cstmt} & ::= & e \mid \mathsf{return}\ e^{?} & \text{(expression and \texttt{return} statements)} \\
 & \mid & \mathsf{goto}\ x \mid x : s & \text{(\texttt{goto} and label)} \\
 & \mid & \mathsf{break} \mid \mathsf{continue} & \text{(\texttt{break} and \texttt{continue})} \\
 & \mid & \{s\} & \text{(block scope)} \\
 & \mid & \overrightarrow{sto}\ \tau\ x := I^{?}\ ;\ s & \text{(local variable declaration)} \\
 & \mid & \mathsf{typedef}\ x := \tau\ ;\ s & \text{(local typedef declaration)} \\
 & \mid & \mathsf{skip} \mid s_1\ ;\ s_2 & \text{(the skip statement and composition)} \\
 & \mid & \mathsf{while}(e)\ s \mid \mathsf{do}\ s\ \mathsf{while}(e) & \text{(\texttt{while} loops)} \\
 & \mid & \mathsf{for}(e_1\ ;\ e_2\ ;\ e_3)\ s & \text{(\texttt{for} loop)} \\
 & \mid & \mathsf{if}\ (e)\ s_1\ \mathsf{else}\ s_2 & \text{(conditional statement)}
\end{array}
$$

The intended semantics of most statements is as expected. Block scopes $\{s\}$ are related to scoping of local variable and typedef declarations. These declarations may be shadowed, but only if a new scope is opened.

Variable declarations with **`extern`** storage bring a global variable into the current scope. These declarations do not have to be in the right order:

```
int f(int x) {
  { extern int x;
    printf("%d\n", x); // x refers to the global variable
  }
  return x;            // x refers to the function parameter
}
int x = 10;
```

Definition 7.1.4. CH$_2$O abstract C declarations *are inductively defined as:*

$$\begin{aligned}
d \in \mathsf{decl} ::=\ & \mathsf{struct}\ \overrightarrow{\tau\, x} \mid \mathsf{union}\ \overrightarrow{\tau\, x} && \text{(struct and union declarations)}\\
\mid\ & \mathsf{enum}\ \overrightarrow{x := e^?} : \tau_\mathsf{i} && \text{(enum declaration)}\\
\mid\ & \mathsf{typedef}\ \tau && \text{(global typedef declaration)}\\
\mid\ & \mathsf{global}\ I^? : \overrightarrow{\mathit{sto}}\ \tau && \text{(global variable declaration)}\\
\mid\ & \mathsf{fun}\ s : \overrightarrow{\mathit{sto}}\ \tau && \text{(function declaration)}
\end{aligned}$$

Definition 7.1.5. CH$_2$O abstract C programs *are defined as:*

$$\Theta \in \mathsf{decls} := \mathsf{list}\ (\mathsf{string} \times \mathsf{decl}).$$

Similar to C source files, a CH$_2$O abstract C program consists of a mixed sequence of declarations:

- The declarations $(s, \mathsf{struct}\ \overrightarrow{\tau\, x})$ and $(u, \mathsf{union}\ \overrightarrow{\tau\, x})$ introduce a struct or union type with fields $\vec{x}$ of type $\vec{\tau}$.
- The declaration $(t, \mathsf{typedef}\ \tau)$ introduces a *typedef* named t that abbreviates τ.
- The declaration $(u, \mathsf{enum}\ \overrightarrow{x := e^?} : \tau_\mathsf{i})$ introduces an enum type named u that designates an integer type τ_i. This declaration furthermore introduces integer constants $\vec{x}$ whose values are specified by optional constant expressions $\vec{e^?}$. The type τ_i is made explicit because it is implementation-defined and may differ for each declared enum type [ISO12, 6.7.2.2p4].
- The declaration $(x, \mathsf{global}\ I^? : \overrightarrow{\mathit{sto}}\ \tau)$ introduces a global variable named x whose value is specified by the initializer $I^?$.
- The declaration $(f, \mathsf{fun}\ s : \overrightarrow{\mathit{sto}}\ \tau)$ introduces a function named f of type τ with body s. An incomplete function should be declared as a global variable without an initializer.

Struct, union and enum types have a different namespace than other declarations. Note that C allows struct, union and enum declarations intermingled with expressions and statements whereas CH$_2$O abstract C does not support that yet.

7.2 Translation into $\mathrm{CH_2O}$ core C

This section describes the translation from $\mathrm{CH_2O}$ abstract C into $\mathrm{CH_2O}$ core C. This translation turns a sequence of declarations $\Theta \in \mathsf{decls}$ into:

- An environment $\Gamma \in \mathsf{env}$ that assigns fields to struct and union names, as well as argument and return types to function names.
- An initial memory $m \in \mathsf{mem}$ that contains the global and static variables, as well as only storage for each string literal.
- A $\mathrm{CH_2O}$ core C program $\delta \in \mathsf{funenv}$ that assigns function bodies to all declared functions.

The translation will fail with an error message in case the given $\mathrm{CH_2O}$ abstract C program is ill-typed, uses undeclared variables, or has other kinds of errors. Type soundness of the translation guarantees that we have $\vdash \Gamma$ and $\Gamma \vdash m$ and $\Gamma, \overline{m} \vdash \delta$ in case the translation succeeds. The translation into $\mathrm{CH_2O}$ core C is defined using a combined error state monad.

Definition 7.2.1. *Given a set of states S and a set of errors E, the* combined error state monad $\mathcal{E}^E_S$ *is defined as:*

$$\mathcal{E}^E_S(A) := S \to E + (A \times S).$$

The monadic operations are defined as expected.

Statefulness is used to keep track of the initial memory among other things. For example, statements may contain static variable declarations that require the initial memory to be extended. Consider the following program:

```
int f() { static int x = 10; return x++; } // x retains its value
int main() { return (f(), f(), f()); }     // returns 12
```

The static variable `x` acts like a global variable that is scoped to the function `f`. The translation of static variable declarations therefore extends the initial memory with an object for the static variable.

Definition 7.2.2. A translation state $\mathcal{S} \in \mathsf{tstate}$ *consists of:*

1. *A finite map* $\mathsf{tag} \to_{\mathsf{fin}} \mathsf{typedecl}$ *of* type declarations*, where:*

$$\mathsf{typedecl} ::= \mathsf{struct}\ \overrightarrow{\tau\, x} \mid \mathsf{union}\ \overrightarrow{\tau\, x} \mid \mathsf{enum}\ \tau_{\mathsf{i}}.$$

2. *An* initial memory $m \in \mathsf{mem}$.
3. *A finite map* $\mathsf{string} \to_{\mathsf{fin}} \mathsf{globaldecl}$ *of* global declarations*, where:*

$$\mathsf{globaldecl} ::= \mathsf{global}\ sto\ o\ \tau\ \beta \mid \mathsf{fun}\ sto\ \vec{\tau}\ \tau\ s^{?} \mid \mathsf{typedef}\ \tau_{\mathsf{p}} \mid \mathsf{enumval}\ \tau_{\mathsf{i}}\ z.$$

The Boolean β denotes whether a global variable has been initialized or not.

Definition 7.2.3. The $\mathrm{CH_2O}$ abstract C translation monad $\mathcal{M}$ *is defined as:*

$$\mathcal{M}(A) := \mathcal{E}^{\mathsf{string}}_{\mathsf{tstate}}(A).$$

In the Coq development we have defined the following monadic translations for each syntactical category of CH$_2$O abstract C:

$$\begin{aligned}
\mathsf{to\_expr} &: \mathsf{localenv} \to \mathsf{cexpr} \to \mathcal{M}(\mathsf{expr} \times \mathsf{lrtype})\\
\mathsf{to\_initexpr} &: \mathsf{localenv} \to \mathsf{type} \to \mathsf{cinit} \to \mathcal{M}(\mathsf{expr})\\
\mathsf{to\_type} &: \mathsf{localenv} \to \mathsf{ctype} \to \mathcal{M}(\mathsf{type})\\
\mathsf{to\_ptrtype} &: \mathsf{localenv} \to \mathsf{ctype} \to \mathcal{M}(\mathsf{ptrtype})\\
\mathsf{to\_stmt} &: \mathsf{localenv} \to \mathsf{type} \to \mathsf{cstmt} \to \mathcal{M}(\mathsf{stmt} \times (\mathsf{bool} \times \mathsf{option\ type}))
\end{aligned}$$

These functions add necessary declarations to the translation state and yield the corresponding CH$_2$O core C construct. The type localenv describes local variable and typedef declarations.

The function to_expr infers the type of an expression and determines whether it is an l-value or r-value. Evaluation of constant expressions [ISO12, 6.6] is performed to convert size expressions of arrays such as **unsigned char**[10 * **sizeof**(**int**)] into integer values, among other things. This conversion is performed using the evaluator for pure expressions (Section 6.9), which is proven sound and complete with respect to the operational semantics (Theorems 6.9.5 on page 129 and 6.9.7 on page 130).

The function to_stmt takes the return type of the C function and inserts casts to make **return** statements well-typed when necessary. Whenever a block scope or local declaration is occurred, it will extend the local environment. Static and external variable declarations extend the initial memory.

In the Coq development we have defined the following functions to translate entire CH$_2$O abstract C programs:

$$\begin{aligned}
\mathsf{alloc\_decls} &: \mathsf{decls} \to \mathcal{M}(1)\\
\mathsf{alloc\_program} &: \mathsf{decls} \to \mathcal{M}(1)
\end{aligned}$$

The function alloc_decls processes a sequence of declarations and adds each declaration to the translation state. In turn, the function alloc_program also checks whether no functions are left without a corresponding body.

Definition 7.2.4. *The functions* $\mathsf{env} : \mathsf{tstate} \to \mathsf{env}$, $\mathsf{mem} : \mathsf{tstate} \to \mathsf{mem}$ *and* $\mathsf{funenv} : \mathsf{tstate} \to \mathsf{funenv}$ *extract the typing environment, initial memory, and function environment from a given translation state.*

Theorem 7.2.5 (Type soundness)**.** *If the translator succeeds, then the results are well-typed. That is, if* $\mathsf{alloc\_program}\ \Theta = \mathsf{return}\ ()\ \mathcal{S}$, *then we have:*

$$\vdash \mathsf{env}\,\mathcal{S} \qquad \text{and} \qquad \mathsf{env}\,\mathcal{S} \vdash \mathsf{mem}\,\mathcal{S} \qquad \text{and} \qquad \mathsf{env}\,\mathcal{S}, \overline{\mathsf{mem}\,\mathcal{S}} \vdash \mathsf{funenv}\,\mathcal{S}.$$

7.3 Machine architectures

The entire CH$_2$O semantics is parameterized by an arbitrary implementation environment that describes implementation-defined attributes such as the sizes and endianness of integers, and the layout of struct and union types (see Definition 3.4.1

on page 48). However, to use the CH$_2$O semantics to execute a concrete program we have to assign concrete values to implementation-defined attributes.

The CH$_2$O interpreter provides various flags that can be used to execute a program using different implementation-defined attributes. These flags are:

1. Big or little endian.
2. ILP32, LLP64 (64 bit Windows) or LP64 (64 bit Unix-derivatives) data model.
3. Signed or unsigned char.

Based on these flags we construct a corresponding implementation environment. The most interesting part of this construction is the layout of types. The alignment and size of each base type is specified by the following table:

	char	short	int	long	long long	Pointers
ILP32	1	2	4	4	8	4
LLP64	1	2	4	4	8	8
LP64	1	2	4	8	8	8

These constants are lifted to full types following standard application binary interfaces (ABI) conventions [MHJM13]. Structs and unions have the alignment of their most strictly aligned field, and each field of a struct or union has the lowest available offset that is correctly aligned.

7.4 The CH$_2$O interpreter

In order to compute the behaviors of a C program, we lift the single step executable semantics $\mathsf{exec} : \mathsf{state} \to \mathcal{P}_{\mathsf{fin}}(\mathsf{state})$ to a function that computes all execution orders of a given program, and to a function that computes a specific execution order of a given program. Executions are represented as streams.

Definition 7.4.1. *The function* $\mathsf{all}_{\Gamma;\delta} : \mathcal{P}_{\mathsf{fin}}(\mathsf{state}) \to (\mathcal{P}_{\mathsf{fin}}(\mathsf{state}) \times \mathcal{P}_{\mathsf{fin}}(\mathsf{state}))^{\omega}$ *computes* the stream of reachable states*. It is coinductively defined as:*

$$\mathsf{all}_{\Gamma;\delta}\ \mathbf{S} := (\mathbf{S}',\ \mathbf{N})\ (\mathsf{all}_{\Gamma;\delta}\ \mathbf{S}')$$

where $\mathbf{S}' := \{S' \mid S' \in \mathsf{exec}_{\Gamma,\delta}\ S, S \in \mathbf{S}\}$ *and* $\mathbf{N} := \{S \mid \mathsf{exec}_{\Gamma,\delta}\ S = \emptyset, S \in \mathbf{S}\}$*.*

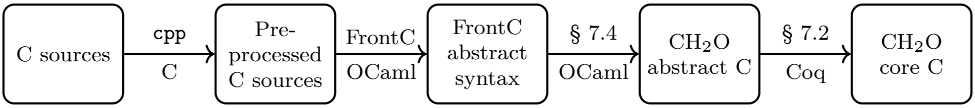
The nth element $(\mathbf{S}, \mathbf{N})$ of the stream $(\mathsf{all}_{\Gamma;\delta}\ \mathbf{I})$ contains the intermediate states $\mathbf{S} \subseteq \mathsf{state}$ and normal forms $\mathbf{N} \subseteq \mathsf{state}$ after n steps starting in initial states $\mathbf{I} \subseteq \mathsf{state}$.

Definition 7.4.2. *Given a selection function* $f : \mathcal{P}_{\mathsf{fin}}^{\neq\emptyset}(\mathsf{state}) \to \mathsf{state}$ *with* $f\,\mathbf{S} \in \mathbf{S}$ *for each* $\emptyset \neq \mathbf{S} \subseteq \mathsf{state}$*, the function* $\mathsf{some}_{\Gamma;\delta}\ f : \mathsf{state} \to (\mathsf{state} + \mathsf{state})^{\omega}$ *computes* a specific execution stream*. It is coinductively defined as:*

$$\mathsf{some}_{\Gamma;\delta}\ f\ S := \begin{cases} S_{\mathbf{r}}\ S_{\mathbf{r}}\ \ldots & \text{if } \mathsf{exec}_{\Gamma,\delta}\ S = \emptyset \\ (f\,\mathbf{S})_{\mathbf{l}}\ (\mathsf{some}_{\Gamma;\delta}\ f\ (f\,\mathbf{S})) & \text{if } \mathbf{S} = \mathsf{exec}_{\Gamma,\delta}\ S \neq \emptyset \end{cases}$$

Given a selection function $f : \mathcal{P}_{\mathsf{fin}}^{\neq\emptyset}(\mathsf{state}) \to \mathsf{state}$ that specifies which redex should be chosen, the stream $(\mathsf{some}_{\Gamma;\delta}\ f\ I)$ represents a trace starting in $I \in \mathsf{state}$. In the disjoint union $(\mathsf{state} + \mathsf{state})$, the left variant is an intermediate state, while the right variant is a normal form. Once in normal form, the stream stays constant.

The translator from CH_2O abstract C into CH_2O core C, the executable semantics, and the above functions are all implemented in Coq. We use Coq's extraction mechanism [Let04] to extract these functions into OCaml. The extracted code is used as part of an interpreter that consists of the following phases:



Since the syntax of CH_2O abstract C is reasonably close to the FrontC abstract syntax, the translation from FrontC abstract syntax trees to CH_2O abstract C is extremely straightforward. If the given program uses a feature that is not supported by CH_2O abstract C, it produces an error message.

The interpreter, called `ch2o`, has three modes: first execution order, random execution order (`-r`), and all possible execution orders (`-t`). Consider:

```
int f(int *p, int z) { return *p = z; }
int main() {
  int x, y = f(&x, 3) + f(&x, 4) + f(&x, 3);
  return x;
}
```

The output of this program using `ch2o -t` is:

```
......+;!|???***%%%%%%****??|||?????|||||!;:::;;!!;::++++--,,,,,,
,,,,,,,,
"" 3
"" 4
```

In the output, the density of the symbols indicate the size of the reachable state sets. For example, "`,`" means 2 states, and "`%`" means between 1025 and 4096 states. The strings `""3` and `""4` indicate the possible behaviors of the program: it either has the empty output and returns `3`, or it has the empty output and returns `4`. Although `x` is modified multiple times in the same expression, this program has defined behavior because function calls have a sequence point.

If a given program has undefined behavior, `ch2o` will explicitly state this undefined behavior. Consider:

```
int main() {
  int x = 10;
  (x = 1) + x;
  return printf("%d\n", x);
}
```

The output of this program using `ch2o -t` is:

```
....,......,-,,
undef
.............,......-+++-,,,,,,,,-::;;;::-,,,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,,,,,-++-,,,,,,,,,,,,,,,.....................
"1\n" 2
```

This program has two possible behaviors. Either the assignment `x = 1` is executed first, in which case reading from `x` has undefined behavior due to a sequence point violation. Alternatively, the assignment is executed after the read, in which case the string `"1\n"` is printed and `2` is returned.

Although C is not confluent, the size of the reachable state set reduces quickly in the absence of function calls. Consider a (simplified) reduction of `*p + *p`:

$$\begin{tikzcd}
\mathbf{S}_0 = \{ & & \texttt{*p + *p} \arrow[dl] \arrow[dr] & & \} \\
\mathbf{S}_1 = \{ & \texttt{10 + *p} \arrow[dr] & & \texttt{*p + 10} \arrow[dl] & \} \\
\mathbf{S}_2 = \{ & & \texttt{20} & & \}
\end{tikzcd}$$

Contrary to λ-calculus and functional programming languages, redexes cannot be duplicated in C. Hence, it is easy to observe that the number of steps on the left hand side is equal to the number of steps on the right hand side.

Consequently, in order to allow for feasible exploration of non-determinism we do not need to keep track of the entire reachable state set but just of the set of reachable states $\mathbf{S}_n$ in exactly n steps. In the Coq development we have implemented $\mathsf{all}_{\Gamma;\delta}$ using verified hashsets to filter out duplicates in our representation of finite sets. The implementation of hashsets in Coq will be described in Chapter 9.

7.5 Benchmarks

The CH$_2$O interpreter makes it possible to test the CH$_2$O semantics against actual C source files. Our interpreter is meant to be an *exploration* tool, intended to explore the implications of the C11 standard, instead of being a debugging tool meant to execute large programs written in C. For exploration purposes, one typically considers small programs that exercise corner cases. This section describes the results of testing our semantics against a small test suite for both defined and undefined behavior, and compares the efficiency of the CH$_2$O interpreter against similar tools.

In order to test the CH$_2$O semantics we need the `printf` function, which prints a string to the standard output and returns the length of the printed string [ISO12, 7.21.6.3]. Since variadic function and I/O are not yet part of the CH$_2$O semantics, we have temporarily implemented `printf` using a trick in the OCaml layer that translates FrontC abstract syntax into CH$_2$O abstract C.

The OCaml layer introduces a separate function declaration for each `printf` call. The introduced function computes the length of the resulting string without actual side-effects. When a call to the introduced function occurs during program execution, the OCaml loop handles the actual print. Consider:

```
int f(int x) { return printf("%d %d\n", x, x*x); }
```

The above excerpt is translated into the following sequence of CH_2O abstract C declarations where i is a unique identifier for the particular `printf`:

$$\begin{aligned}(\texttt{printf\_}i,\ \mathsf{fun}\,(\text{compute the length of ``}x_1\ \ x_2\texttt{\textbackslash n}\text{''}) :\ [\,\mathsf{int}\,x_1,\ \mathsf{int}\,x_2\,] \to \mathsf{int}),\\ (\texttt{f},\ \mathsf{fun}\,(\mathsf{return}\ \texttt{printf\_}i(x,\ x * x)) :\ \mathsf{int}\,x \to \mathsf{int})\end{aligned}$$

The function $\texttt{printf\_}i$ does not have side-effects in terms of the CH_2O semantics. Whenever the state $\mathbf{S}(\mathcal{P},\ \overline{\mathsf{call}}\,\texttt{printf\_}i\,\vec{v},\ m)$ occurs in the stream of intermediate states, the OCaml loop will perform the actual print. We furthermore made a small modification to the interpreter, as we store the printed results with the CH_2O states. This has not affected the CH_2O core semantics.

We have tested the CH_2O semantics against a small test suite for both defined and undefined behavior[1]. This test suite is partially based on subtle examples from the C standard [ISO12], C defect reports [ISO], and tests by Ellison and Roşu [ER12b]. For the tests by Ellison and Roşu we have obtained the following results:

	Agree	Disagree	Unsupported
Defined behavior	181	9 undefined in CH_2O	65
Undefined behavior	123	0 defined in CH_2O	66

Unsupported programs make use of features not supported by CH_2O like floats, bitfields, threads, the **`register`**, **`const`** or **`volatile`** keyword.

The CH_2O semantics assigns undefined behavior to eight tests for defined behavior. The reasons for these differences between Ellison and Roşu and us include:

- Casting an integer to a signed type that cannot represent the result of the cast yields implementation-defined or undefined behavior in CH_2O whereas Ellison and Roşu assign a specific value. The C11 standard states that either a signal is raised, or that the resulting value is implementation-defined [ISO12, 6.3.1.3].
- CH_2O assigns undefined behavior to corner cases related to the sequence point restriction that have defined behavior in the semantics of Ellison and Roşu. We have discussed these differences in Section 6.4.

Since we have formally proven many properties of our language as part of published papers [KW13, Kre13, Kre14a, Kre14b] before we implemented the interpreter, it comes to no surprise that we we did not discover many fundamental bugs. Bugs that we discovered were mostly related to forgotten implicit casts, too lenient restrictions on integer operations, or absent checks in the translation to CH_2O core C.

Although our executable semantics is a naive implementation that enumerates all paths through the operational semantics, and reuses our inefficient memory model, it is still efficient enough explore the entire state space of small programs. It is much

[1] Available online at `https://github.com/robbertkrebbers/ch2o/tree/master/tests/`.

faster than the semantics of Ellison and Roşu. For example, calculating all execution orders of the following seemingly innocent but very non-deterministic program takes 7s using our semantics and 16m using the semantics of Ellison and Roşu.

```
int x = 10, *p = &x;
return *p + *p + *p + *p + *p + *p;
```

This major difference is explained by the fact that we avoid the overhead of a rewriting framework like the $\mathbb{K}$-framework. Our interpreter is a functional program written in Coq, extracted to OCaml using Coq's extraction mechanism, and then compiled to native code using the optimizing OCaml compiler.

The executable semantics of CompCert is also written in Coq and extracted to OCaml. We compared the efficiency on the programs display in Figure 7.1:

	CH$_2$O	CompCert
`fib_5.c`	0.6s	64.0s
`fib_6.c`	10.4s	241.0m
`subexprs_6.c`	7.4s	0.7s
`subexprs_4_100.c`	6.6s	2.7s
`subexprs_4_1000.c`	60.5s	43.4s
`big_array_1000.c`, first execution order	7.5s	0.0s
`big_array_1000.c`, all execution orders	58.1s	0.7s

The differences in efficiency can be explained as follows:

- Program states of CH$_2$O are bigger than those of CompCert. This is because we tag each individual byte in memory with a rich permission whereas CompCert tags each byte with a much coarser permission. Bigger program states seem to make CH$_2$O overall slower.
- Due to the complex nature of the CH$_2$O memory model, our interpreter is slower than CompCert's on programs with large arrays such as `big_array_1000.c`. We represent arrays using lists, whereas CompCert uses maps that have operations with logarithmic time complexity.
- CompCert keeps track of the entire reachable state set, whereas we only keep track of the reachable states after n steps. Since redexes cannot be duplicated in C (unlike languages based on λ-calculus), our approach is a better choice.

  The program `fib_`n`.c` is an example where our approach performs an order of magnitude better. Bigger values for n in `subexprs_4_`n`.c` introduce more non-determinism and bring the timings of CH$_2$O closer to those of CompCert.

There are many directions for future research to improve the efficiency. First of all, our executable semantics uses a memory model that is defined with clarity instead of efficiency in mind. It would therefore be useful to define an isomorphic version of our memory model that uses more efficient data structures. Since we use a proof assistant, both versions can coexist and can be proven to be isomorphic.

Second of all, in both CompCert and CH$_2$O, non-constant expressions without side-effects are executed non-deterministically as well as in small-steps. It would be interesting to investigate whether we could use a deterministic evaluator to reduce

The files `fib_n.c` for any $n \in \mathbb{N}$:

```
int x;
int set_x(int y) { x = 1; return y; }
int two_unspec() { x = 0; return x + set_x(1); }
int add_zero(int y) { x = 0; return y - x + set_x(0); }
int fib(int y) {
  if (y > 3) return fib(y - 2) + add_zero(fib(y - 1));
  else if (y == 3) return two_unspec();
  else return 1;
}
int main() { printf("Fibonacci n = %d\n", fib(n)); return 0; }
```

(By Jones, see `http://lists.cs.uiuc.edu/pipermail/c-semantics/2011-June/000034.html`)

The file `subexprs_6.c`:

```
int main() {
  int x = 10, *p = &x;
  return *p + *p + *p + *p + *p + *p;
}
```

The files `subexprs_4_n.c` for any $n \in \mathbb{N}$:

```
int main() {
  int x = 10, *p = &x;
  for (int i = 0; i < n; i++) *p + *p + *p + *p;
  return 0;
}
```

The file `big_array_1000.c`:

```
int main() {
  int a[1000];
  for (int i = 0; i < 1000; i++) a[i] = i;
  return a[10];
}
```

Figure 7.1: Overview of benchmark programs.

subexpressions without side-effects. Confluence results for other classes of C expressions as proven by Norrish [Nor99] may be helpful too.

Third of all, one may apply ordinary program transformations that preserve all behaviors (that is, do not remove undefined behaviors).

Chapter 8

Separation logic

This chapter describes an axiomatic semantics in the form of a separation logic for $\mathrm{CH_2O}$ core C, which has been proven sound with respect to the operational semantics (Theorem 8.6.13). It correctly deals with features of C that are not considered by others, such as non-deterministic expressions evaluation in the presence of side-effects and block scope local variables in the presence of non-local control flow. Our treatment of the former is inspired by concurrent separation logic [O'H04].

Judgments of a conventional axiomatic semantics for partial program correctness are Hoare triples $\{P\}\, s\, \{Q\}$ where s is a statement, and P and Q are *assertions* called the pre- and postcondition. The intuitive reading of a Hoare triple $\{P\}\, s\, \{Q\}$ is: if P holds for the memory before executing s, and execution of s terminates, then Q holds for the memory afterwards. As is common in separation logic [ORY01], our Hoare triples also ensure the absence of undefined behavior.

The axiomatic semantics for $\mathrm{CH_2O}$ core C has separate judgments for expressions (Definition 8.4.4) and statements (Definition 8.5.3). Expression judgments are triples $\vdash_{\Gamma,\delta} \{P\}\, e\, \{Q\}$ where $\Gamma \in \mathsf{env}$ is a type environment and $\delta \in \mathsf{funenv}$ contains the bodies of all declared functions. The postcondition Q is a function from values to assertions to account for the fact that an expression not only performs side-effects, but primarily yields a value. The judgment $\vdash_{\Gamma,\delta} \{P\}\, e\, \{Q\}$ ensures that if e yields an address or value $\nu \in \mathsf{lrval}$, then $Q\,\nu$ holds afterwards.

Statement judgments are sextuples $R, J, T \vdash_{\Gamma,\delta} \{P\}\, s\, \{Q\}$. The right hand side is a traditional Hoare triple, and the environments deal with non-local control:

- The environment R specifies the conditions for **return**s.
- The environment J specifies the conditions for **goto**s.
- The environment T specifies the conditions for **break** and **continue**.

We define the assertions of our axiomatic semantics through a shallow embedding (Sections 8.2 and 8.3). In order to do so, we show that the $\mathrm{CH_2O}$ memory model is a separation algebra (Section 8.1). We then present the inference rules for expressions (Section 8.4) and statements (Section 8.5) and prove soundness with respect to the operational semantics (Section 8.6). Finally, we extend our separation logic with some additional features to make it more suitable for program verification (Section 8.7) and conclude the chapter by verifying the C source code of a version of Euclid's algorithm for computing the greatest common divisor (Section 8.8).

In the Coq development, we have also defined the judgments $\vdash_{\Gamma,\delta} \{P\}\, e\, \{Q\}$ and $R, J, T \vdash_{\Gamma,\delta} \{P\}\, s\, \{Q\}$ through a shallow embedding. For clarity of presentation, we explicitly present these judgments as an inference system in this thesis.

8.1 The memory as a separation algebra

In this section we show that the CH$_2$O memory model is a separation algebra, and that the separation algebra operations interact appropriately with the memory operations that we have defined in Chapter 5.

Section 8.2 will represent the assertions of our separation logic through a shallow embedding, which roughly means that assertions $P, Q : \mathsf{mem} \to \mathsf{Prop}$ are predicates over memory states[1]. Recall that using a shallow embedding the *separating conjunction* $*$ of separation logic is defined as follows [ORY01]:

$$P * Q := \lambda m\,.\, \exists m_1\, m_2\,.\, m = m_1 \cup m_2 \wedge m_1 \perp m_2 \wedge P\, m_1 \wedge Q\, m_2.$$

The separating conjunction $P * Q$ allows one to subdivide the memory into two disjoint parts such that P holds in one part and Q in the other part. Subdivision is described using the separation algebra operation $\cup : \mathsf{mem} \to \mathsf{mem} \to \mathsf{mem}$. This operation should enjoy two important properties:

- It should allow one to subdivide individual objects into multiple copies of the same object with lower permissions. In terms of the singleton assertion this means (see Lemma 8.3.6 for the actual result):

  $$(a \xmapsto{\gamma_1 \cup \gamma_2} v) \quad \leftrightarrow \quad (a \xmapsto{\gamma_1} v) * (a \xmapsto{\gamma_2} v)$$

  The *singleton assertion* $a \xmapsto{\gamma} v$ denotes that the memory consists of exactly one object with value v at address a that has permission γ (Definition 8.3.3). The $\cup$ operation corresponds to the separation algebra on permissions, which we have defined in Chapter 4.

- It should allow one to subdivide compound objects such as arrays, structs and unions into multiple parts. In terms of the singleton assertion this means for the case of arrays (see Lemma 8.3.5 for the actual result):

  $$(a \xmapsto{\gamma} \mathsf{array}\,[\, v_0, \ldots, v_{n-1}\,]) \quad \leftrightarrow \quad (a[0] \xmapsto{\gamma} v_0) * \cdots * (a[n-1] \xmapsto{\gamma} v_{n-1}).$$

In order to define the separation algebra relations and operations on memories, we first define these on memory trees. Memory trees do not form a separation algebra themselves due to the absence of a unique $\emptyset$ element (memory trees have a distinct identity element new_Γ^τ for each type τ, see Definition 5.4.9 on page 76). The separation algebra of memories is then defined by lifting the definitions on memory trees to memories (which are basically finite functions to memory trees).

[1] The actual definition is more complicated because we also have to deal with typing environments and the stack, see Definition 8.2.1.

Definition 8.1.1. *The predicate* $\mathsf{valid} : \mathsf{mtree} \to \mathsf{Prop}$ *is inductively defined as:*

$$\frac{\mathsf{valid}\ \vec{\mathbf{b}}}{\mathsf{valid}\ (\mathsf{base}_{\tau_\mathsf{b}}\ \vec{\mathbf{b}})} \qquad \frac{\mathsf{valid}\ \vec{w}}{\mathsf{valid}\ (\mathsf{array}_\tau\ \vec{w})} \qquad \frac{\mathsf{valid}\ \vec{w} \qquad \mathsf{valid}\ \vec{\vec{\mathbf{b}}}}{\mathsf{valid}\ (\mathsf{struct}_t\ \overrightarrow{w\vec{\mathbf{b}}})}$$

$$\frac{\mathsf{valid}\ w \qquad \mathsf{valid}\ \vec{\mathbf{b}} \qquad \neg\mathsf{unmapped}\ (\overline{w}\,\vec{\mathbf{b}})}{\mathsf{valid}\ (\mathsf{union}_t\ (i, w, \vec{\mathbf{b}}))} \qquad \frac{\mathsf{valid}\ \vec{\mathbf{b}}}{\mathsf{valid}\ (\overline{\mathsf{union}_t}\ \vec{\mathbf{b}})}$$

Fact 8.1.2. *If* $\Gamma, \Delta \vdash w : \tau$, *then* $\mathsf{valid}\ w$.

The valid predicate specifies the subset of memory trees on which the separation algebra structure is defined. The definition basically lifts the valid predicate from the leaves to the trees. The side-condition $\neg\mathsf{unmapped}\ (\overline{w}\,\vec{\mathbf{b}})$ on $\mathsf{union}_t\ (i, w, \vec{\mathbf{b}})$ memory trees ensures canonicity, unions whose permissions are unmapped cannot be accessed and are thus kept in unspecified variant. Unmapped unions $\overline{\mathsf{union}_t}\ \vec{\mathbf{b}}$ can be combined with other unions using $\cup$. The rationale for doing so will become clear in Section 8.3 where we define the singleton assertion.

Definition 8.1.3. *The relation* $\perp : \mathsf{mtree} \to \mathsf{mtree} \to \mathsf{Prop}$ *is inductively defined as:*

$$\frac{\vec{\mathbf{b}}_1 \perp \vec{\mathbf{b}}_2}{\mathsf{base}_{\tau_\mathsf{b}}\ \vec{\mathbf{b}}_1 \perp \mathsf{base}_{\tau_\mathsf{b}}\ \vec{\mathbf{b}}_2} \qquad \frac{\vec{w}_1 \perp \vec{w}_2}{\mathsf{array}_\tau\ \vec{w}_1 \perp \mathsf{array}_\tau\ \vec{w}_2} \qquad \frac{\vec{w}_1 \perp \vec{w}_2 \qquad \vec{\vec{\mathbf{b}}}_1 \perp \vec{\vec{\mathbf{b}}}_2}{\mathsf{struct}_t\ \overrightarrow{w_1\vec{\mathbf{b}}_1} \perp \mathsf{struct}_t\ \overrightarrow{w_2\vec{\mathbf{b}}_2}}$$

$$\frac{w_1 \perp w_2 \qquad \vec{\mathbf{b}}_1 \perp \vec{\mathbf{b}}_2 \qquad \neg\mathsf{unmapped}\ (\overline{w_1}\,\vec{\mathbf{b}}_1) \qquad \neg\mathsf{unmapped}\ (\overline{w_2}\,\vec{\mathbf{b}}_2)}{\mathsf{union}_t\ (i, w_1, \vec{\mathbf{b}}_1) \perp \mathsf{union}_t\ (i, w_2, \vec{\mathbf{b}}_2)}$$

$$\frac{\vec{\mathbf{b}}_1 \perp \vec{\mathbf{b}}_2}{\overline{\mathsf{union}_t}\ \vec{\mathbf{b}}_1 \perp \overline{\mathsf{union}_t}\ \vec{\mathbf{b}}_2} \qquad \frac{\overline{w_1}\,\vec{\mathbf{b}}_1 \perp \vec{\mathbf{b}}_2 \quad \mathsf{valid}\ w_1 \quad \neg\mathsf{unmapped}\ (\overline{w_1}\,\vec{\mathbf{b}}_1) \quad \mathsf{unmapped}\ \vec{\mathbf{b}}_2}{\mathsf{union}_t\ (i, w_1, \vec{\mathbf{b}}_1) \perp \overline{\mathsf{union}_t}\ \vec{\mathbf{b}}_2}$$

$$\frac{\vec{\mathbf{b}}_1 \perp \overline{w_2}\,\vec{\mathbf{b}}_2 \qquad \mathsf{valid}\ w_2 \qquad \mathsf{unmapped}\ \vec{\mathbf{b}}_1 \qquad \neg\mathsf{unmapped}\ (\overline{w_2}\,\vec{\mathbf{b}}_2)}{\overline{\mathsf{union}_t}\ \vec{\mathbf{b}}_1 \perp \mathsf{union}_t\ (i, w_2, \vec{\mathbf{b}}_2)}$$

Definition 8.1.4. *The operation* $\cup : \mathsf{mtree} \to \mathsf{mtree} \to \mathsf{mtree}$ *is defined as:*

$$\begin{aligned}
\mathsf{base}_{\tau_\mathsf{b}}\ \vec{\mathbf{b}}_1 \cup \mathsf{base}_{\tau_\mathsf{b}}\ \vec{\mathbf{b}}_2 \quad &:= \mathsf{base}_{\tau_\mathsf{b}}\ (\vec{\mathbf{b}}_1 \cup \vec{\mathbf{b}}_2)\\
\mathsf{array}_\tau\ \vec{w}_1 \cup \mathsf{array}_\tau\ \vec{w}_2 \quad &:= \mathsf{array}_\tau\ (\vec{w}_1 \cup \vec{w}_2)\\
\mathsf{struct}_t\ \overrightarrow{w_1\vec{\mathbf{b}}_1} \cup \mathsf{struct}_t\ \overrightarrow{w_2\vec{\mathbf{b}}_2} \quad &:= \mathsf{struct}_t\ (\overrightarrow{w_1\vec{\mathbf{b}}_1} \cup \overrightarrow{w_2\vec{\mathbf{b}}_2})\\
\mathsf{union}_t\ (i, w_1, \vec{\mathbf{b}}_1) \cup \mathsf{union}_t\ (i, w_2, \vec{\mathbf{b}}_2) \quad &:= \mathsf{union}_t\ (i, w_1 \cup w_2, \vec{\mathbf{b}}_1 \cup \vec{\mathbf{b}}_2)\\
\overline{\mathsf{union}_t}\ \vec{\mathbf{b}}_1 \cup \overline{\mathsf{union}_t}\ \vec{\mathbf{b}}_1 \quad &:= \overline{\mathsf{union}_t}\ (\vec{\mathbf{b}}_1 \cup \vec{\mathbf{b}}_2)\\
\mathsf{union}_t\ (i, w_1, \vec{\mathbf{b}}_1) \cup \overline{\mathsf{union}_t}\ \vec{\mathbf{b}}_2 \quad &:= \mathsf{union}_t\ (i, w_1, \vec{\mathbf{b}}_1) \mathbin{\hat{\cup}} \vec{\mathbf{b}}_2\\
\overline{\mathsf{union}_t}\ \vec{\mathbf{b}}_1 \cup \mathsf{union}_t\ (i, w_2, \vec{\mathbf{b}}_2) \quad &:= \mathsf{union}_t\ (i, w_2, \vec{\mathbf{b}}_2) \mathbin{\hat{\cup}} \vec{\mathbf{b}}_1
\end{aligned}$$

In the last two clauses, $w \mathbin{\hat{\cup}} \vec{\mathbf{b}}$ *is a modified version of the memory tree* w *in which the elements on the leaves of* w *are zipped with* $\vec{\mathbf{b}}$ *using the* $\cup$ *operation on permission annotated bits (see Definitions 5.4.16 on page 79 and 4.4.3 on page 57).*

The definitions of valid, $\perp$ and $\cup$ on memory trees satisfy all laws of a separation algebra (see Definition 4.2.1 on page 52) apart from those involving $\emptyset$. We prove the cancellation law explicitly as it involves the aforementioned side-conditions on unions.

Lemma 8.1.5. *If $w_3 \perp w_1$ and $w_3 \perp w_2$ then:*

$$w_3 \cup w_1 = w_3 \cup w_2 \quad \text{implies} \quad w_1 = w_2.$$

Proof. By induction on the derivations $w_3 \perp w_1$ and $w_3 \perp w_2$. We consider one case:

$$\frac{\begin{array}{c}\mathsf{union}_t\,(i, w_3, \vec{\mathbf{b}}_3) \perp \mathsf{union}_t\,(i, w_1, \vec{\mathbf{b}}_1) \qquad \mathsf{union}_t\,(i, w_3, \vec{\mathbf{b}}_3) \perp \overline{\mathsf{union}_t}\,\vec{\mathbf{b}}_2 \\ \mathsf{union}_t\,(i, w_3, \vec{\mathbf{b}}_3) \cup \mathsf{union}_t\,(i, w_1, \vec{\mathbf{b}}_1) = \mathsf{union}_t\,(i, w_3, \vec{\mathbf{b}}_3) \cup \overline{\mathsf{union}_t}\,\vec{\mathbf{b}}_2\end{array}}{\mathsf{union}_t\,(i, w_1, \vec{\mathbf{b}}_1) = \overline{\mathsf{union}_t}\,\vec{\mathbf{b}}_2}$$

Here, we have $\overline{w_3}\,\vec{\mathbf{b}}_3 \cup \overline{w_1}\,\vec{\mathbf{b}}_1 = \overline{w_3}\,\vec{\mathbf{b}}_3 \cup \vec{\mathbf{b}}_2$ by assumption, and therefore $\overline{w_1}\,\vec{\mathbf{b}}_1 = \vec{\mathbf{b}}_2$ by the cancellation law of a separation algebra. However, by assumption we also have $\neg\mathsf{unmapped}\,(\overline{w_1}\,\vec{\mathbf{b}}_1)$ and $\mathsf{unmapped}\,\vec{\mathbf{b}}_2$, which contradicts $\overline{w_1}\,\vec{\mathbf{b}}_1 = \vec{\mathbf{b}}_2$. □

Definition 8.1.6. *The* separation algebra of memories *is defined as:*

$$\begin{aligned}\mathsf{valid}\,m &:= \forall o\, w\, \mu \,.\, m\, o = (w,\, \mu) \to (\mathsf{valid}\,w \text{ and not } \overline{w} \text{ all } (\emptyset,\, \lightning))\\ m_1 \perp m_2 &:= \forall o \,.\, P\, m_1\, m_2\, o\\ m_1 \cup m_2 &:= \lambda o \,.\, f\, m_1\, m_2\, o\end{aligned}$$

$P : \mathsf{mem} \to \mathsf{mem} \to \mathsf{index} \to \mathsf{Prop}$ *and* $f : \mathsf{mem} \to \mathsf{mem} \to \mathsf{index} \to \mathsf{option\ mtree}$ *are defined by case analysis on $m_1\, o$ and $m_2\, o$:*

$m_1\, o$	$m_2\, o$	$P\, m_1\, m_2\, o$	$f\, m_1\, m_2\, o$
$(w_1,\, \mu)$	$(w_1,\, \mu)$	$w_1 \perp w_2$, not $\overline{w_1}$ all $(\emptyset,\, \lightning)$ and not $\overline{w_2}$ all $(\emptyset,\, \lightning)$	$(w_1 \cup w_2,\, \mu)$
$(w_1,\, \mu)$	$\perp$	$\mathsf{valid}\,w_1$ and not $\overline{w_1}$ all $(\emptyset,\, \lightning)$	$(w_1,\, \mu)$
$\perp$	$(w_2,\, \mu)$	$\mathsf{valid}\,w_2$ and not $\overline{w_2}$ all $(\emptyset,\, \lightning)$	$(w_2,\, \mu)$
τ_1	$\perp$	True	τ_1
$\perp$	τ_2	True	τ_2
$\perp$	$\perp$	True	$\perp$
otherwise		False	$\perp$

The definitions of the omitted relations and operations are as expected.

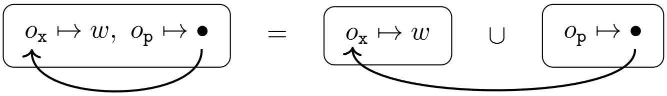
The emptiness conditions ensure canonicity. Objects that solely consist of indeterminate bits with $\emptyset$ permission are meaningless and should not be kept at all. These conditions are needed for cancellativity.

Fact 8.1.7. *If $\Gamma, \Delta \vdash m$, then* $\mathsf{valid}\,m$.

Lemma 8.1.8. *If $m_1 \perp m_2$, then:*

$$\Gamma, \Delta \vdash m_1 \cup m_2 \qquad \text{iff} \qquad \Gamma, \Delta \vdash m_1 \text{ and } \Gamma, \Delta \vdash m_2.$$

Notice that the memory typing environment Δ is not subdivided among m_1 and m_2. Consider the memory state corresponding to **int** `x = 10, *p = &x`:


$o_{\mathtt{x}} \mapsto w,\ o_{\mathtt{p}} \mapsto \bullet$
=
$o_{\mathtt{x}} \mapsto w$
$\cup$
$o_{\mathtt{p}} \mapsto \bullet$

Here, w is the memory tree that represents the integer value 10. The pointer on the right hand side is well-typed in the memory environment $\overline{o_{\mathtt{x}} \mapsto w,\ o_{\mathtt{p}} \mapsto \bullet}$ of the whole memory, but not in $\overline{o_{\mathtt{p}} \mapsto \bullet}$.

We prove some essential properties about the interaction between the separation algebra operations and the memory operations. These properties will be used in the soundness proof of our separation logic in Section 8.6.

Lemma 8.1.9 (Preservation of lookups). *If $\Gamma, \Delta \vdash m_1$ and $m_1 \subseteq m_2$, then:*

$$\begin{array}{rcl} m_1\langle a\rangle_\Gamma = v & \text{implies} & m_2\langle a\rangle_\Gamma = v \\ \mathsf{writable}_\Gamma\, a\, m_1 & \text{implies} & \mathsf{writable}_\Gamma\, a\, m_2 \end{array}$$

The relation $\subseteq$ is part of a separation algebra (see Definition 4.2.1 on page 52). We have $m_1 \subseteq m_2$ iff there is an m_3 with $m_1 \perp m_3$ and $m_2 = m_1 \cup m_3$.

Lemma 8.1.10 (Preservation of disjointness). *If $\Gamma, \Delta \vdash m$ then:*

$$\begin{array}{ll} m \leq_\perp \mathsf{force}_\Gamma\, a\, m & \text{if } \Gamma, \Delta \vdash a : \tau \text{ and } m\langle a\rangle_\Gamma \neq \perp \\ m \leq_\perp m\langle a := v\rangle_\Gamma & \text{if } \Gamma, \Delta \vdash a : \tau \text{ and } \mathsf{writable}_\Gamma\, a\, m \\ m \leq_\perp \mathsf{lock}_\Gamma\, a\, m & \text{if } \Gamma, \Delta \vdash a : \tau \text{ and } \mathsf{writable}_\Gamma\, a\, m \\ m \leq_\perp \mathsf{unlock}\, \Omega\, m & \text{if } \Omega \subseteq \mathsf{locks}\, m \end{array}$$

The relation $\leq_\perp$ is defined in Definition 4.6.3 on page 59. If $m \leq_\perp m'$, then each memory that is disjoint to m is also disjoint to m'.

As a corollary of the previous lemma and Fact 4.6.4 on page 60 we obtain that $m_1 \perp m_2$ implies disjointness of the memory operations:

$$\begin{array}{cc} \mathsf{force}_\Gamma\, a\, m_1 \perp m_2 & m_1\langle a := v\rangle_\Gamma \perp m_2 \\ \mathsf{lock}_\Gamma\, a\, m_1 \perp m_2 & \mathsf{unlock}\, \Omega\, m_1 \perp m_2 \end{array}$$

Lemma 8.1.11 (Unions distribute). *If $\Gamma, \Delta \vdash m$ and $m_1 \perp m_2$ then:*

$$\begin{array}{rcll} \mathsf{force}_\Gamma\, a\, (m_1 \cup m_2) &=& \mathsf{force}_\Gamma\, a\, m_1 \cup m_2 & \text{if } \Gamma, \Delta \vdash a : \tau \text{ and } m_1\langle a\rangle_\Gamma \neq \perp \\ (m_1 \cup m_2)\langle a := v\rangle_\Gamma &=& m_1\langle a := v\rangle_\Gamma \cup m_2 & \text{if } \Gamma, \Delta \vdash \{a, v\} : \tau \text{ and } \mathsf{writable}_\Gamma\, a\, m_1 \\ \mathsf{lock}_\Gamma\, a\, (m_1 \cup m_2) &=& \mathsf{lock}_\Gamma\, a\, m_1 \cup m_2 & \text{if } \Gamma, \Delta \vdash a : \tau \text{ and } \mathsf{writable}_\Gamma\, a\, m_1 \\ \mathsf{unlock}\, \Omega\, (m_1 \cup m_2) &=& \mathsf{unlock}\, \Omega\, m_1 \cup m_2 & \text{if } \Omega \subseteq \mathsf{locks}\, m_1 \end{array}$$

Memory trees and memories can be generalized to contain elements of an arbitrary separation algebra as leaves instead of just permission annotated bits [Kre14b]. These generalized memories form a functor that lifts the separation algebra structure on the leaves to entire trees. We have taken this approach in the Coq development, but for brevity's sake, we have refrained from doing so in this thesis.

8.2 Separation logic assertions

We use a *shallow embedding* [WN04] to represent the assertions of our separation logic. A shallow embedding avoids the indirection of first having to define a syntax and then a semantic interpretation of that syntax. It thus eliminates some tedious work, especially in the context of our Coq development. It furthermore makes it easy to introduce new connectives, as we will do throughout this chapter.

Definition 8.2.1. Assertions $P, Q \in \mathsf{assert}$ *are predicates:*

$$P, Q \in \mathsf{env} \to \mathsf{memenv} \to \mathsf{funenv} \to \mathsf{stack} \to \mathsf{mem} \to \mathsf{Prop}$$

that are closed under weakening of typing environments, forwardness of memory typing environments and weakening of function environments. Formally, that means:

$$P\,\Gamma\,\Delta\,\delta\,\rho\,m \quad \text{implies} \quad P\,\Gamma'\,\Delta'\,\delta'\,\rho\,m$$

for any $\Gamma \subseteq \Gamma'$, $\Delta \Rightarrow \Delta'$ and $\delta \subseteq \delta'$ with $\vdash \Gamma'$, $\Delta' \vdash \rho$ and $\Gamma', \Delta' \vdash m$. The relation $\Rightarrow$ of forwardness of memory typing environments is defined in Definition 6.6.8 on page 122.

Assertions do not merely ensure that a property is true for a given typing environment Γ, memory typing environment Δ and function environment δ, but also for any extensions $\Gamma \subseteq \Gamma'$, $\Delta \Rightarrow \Delta'$ and $\delta \subseteq \delta'$. This is closely related to an intuitionistic Kripke semantics [Kri65]. A Kripke semantics is commonly used in higher-order separation logic with step-indexed assertions, which we discuss in Section 8.7.

Closure under weakening of typing environments Γ ensures that an assertion remains true if the program is extended with additional struct, union or function declarations. Closure under weakening of memory typing environments Δ ensures that an assertion remains true during program execution (which may cause the memory typing environment Δ to grow, see Theorem 6.6.13 on page 123).

Definition 8.2.2. An assertion P entails Q*, notation $P \models_{\Gamma,\delta} Q$, if:*

$$P\,\Gamma'\,\Delta\,\delta'\,\rho\,m \quad \text{implies} \quad Q\,\Gamma'\,\Delta\,\delta'\,\rho\,m$$

for any $\Gamma \subseteq \Gamma'$, Δ, $\delta \subseteq \delta'$, ρ and m with $\vdash \Gamma'$ and $\Delta \vdash \rho$ and $\Gamma', \Delta \vdash m$.

Notation 8.2.3. *We let $P \equiv_{\Gamma,\delta} Q$ denote $P \models_{\Gamma,\delta} Q$ and $Q \models_{\Gamma,\delta} P$.*

Definition 8.2.4. *The logical connectives are defined as:*

$$\begin{aligned}
\mathsf{True} &:= \lambda\,\Gamma\,\Delta\,\delta\,\rho\,m\,.\,\mathsf{True}\\
\mathsf{False} &:= \lambda\,\Gamma\,\Delta\,\delta\,\rho\,m\,.\,\mathsf{False}\\
P \to Q &:= \lambda\,\Gamma\,\Delta\,\delta\,\rho\,m\,.\,\forall \Gamma'\,\Delta'\,\delta'\,.\,\Gamma \subseteq \Gamma' \;\to\; \Delta \Rightarrow \Delta' \;\to\; \delta \subseteq \delta'\\
&\qquad\qquad\qquad\qquad\qquad\qquad \vdash \Gamma' \;\to\; \Gamma', \Delta' \vdash \delta' \;\to\; \Gamma', \Delta' \vdash m \;\to\\
&\qquad\qquad\qquad\qquad\qquad\qquad P\,\Gamma'\,\Delta'\,\delta'\,\rho\,m \;\to\; Q\,\Gamma'\,\Delta'\,\delta'\,\rho\,m\\
P \wedge Q &:= \lambda\,\Gamma\,\Delta\,\delta\,\rho\,m\,.\,P\,\Gamma\,\Delta\,\delta\,\rho\,m \;\wedge\; Q\,\Gamma\,\Delta\,\delta\,\rho\,m\\
P \vee Q &:= \lambda\,\Gamma\,\Delta\,\delta\,\rho\,m\,.\,P\,\Gamma\,\Delta\,\delta\,\rho\,m \;\vee\; Q\,\Gamma\,\Delta\,\delta\,\rho\,m\\
\forall x\,.\,P\,x &:= \lambda\,\Gamma\,\Delta\,\delta\,\rho\,m\,.\,\forall x\,.\,(P\,x)\,\Gamma\,\Delta\,\delta\,\rho\,m\\
\exists x\,.\,P\,x &:= \lambda\,\Gamma\,\Delta\,\delta\,\rho\,m\,.\,\exists x\,.\,(P\,x)\,\Gamma\,\Delta\,\delta\,\rho\,m
\end{aligned}$$

We often lift these connectives to functions $A \to \mathsf{assert}$. For example, given $P, Q : A \to \mathsf{assert}$ we write $P \wedge Q$ instead of $\lambda z \,.\, P\, z \wedge Q\, z$.

We now define the separation logic connectives [ORY01]. The assertion emp asserts that the memory is empty. The *separating conjunction* $P * Q$ asserts that the memory can be subdivided into two disjoint parts such that P holds in one part and Q in the other (due to the use of permissions, these parts may partially overlap as long as their permissions are disjoint and their values agree). The *magic wand* $P \mathrel{-\!\!*} Q$ assert that if a given memory is extended with a disjoint part in which P holds, then Q holds in the extended memory.

Definition 8.2.5. *The* connectives of separation logic *are defined as:*

$$
\begin{aligned}
\ulcorner A \urcorner &:= \lambda \Gamma\, \Delta\, \delta\, \rho\, m \,.\, m = \emptyset \ \wedge\ A \\
\mathsf{emp} &:= \ulcorner \mathsf{True} \urcorner \\
P * Q &:= \lambda \Gamma\, \Delta\, \delta\, \rho\, m \,.\, \exists m_1\, m_2 \,.\, m = m_1 \cup m_2 \ \wedge\ m_1 \perp m_2 \ \wedge \\
&\qquad\qquad\qquad\qquad\qquad\quad P\, \Gamma\, \Delta\, \delta\, \rho\, m_1 \ \wedge\ Q\, \Gamma\, \Delta\, \delta\, \rho\, m_2 \\
P \mathrel{-\!\!*} Q &:= \lambda \Gamma\, \Delta\, \delta\, \rho\, m_2 \,.\, \forall \Gamma'\, \Delta'\, \delta'\, m_1 \,.\, \Gamma \subseteq \Gamma' \ \to\ \Delta \Rightarrow \Delta' \ \to\ \delta \subseteq \delta' \ \to \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdash \Gamma' \ \to\ \Gamma', \Delta' \vdash \delta' \ \to\ m_1 \perp m_2 \ \to \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Gamma', \Delta' \vdash m_1 \ \to\ \Gamma', \Delta' \vdash m_2 \ \to \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad P\, \Gamma'\, \Delta'\, \delta'\, \rho\, m_1 \ \to\ Q\, \Gamma'\, \Delta'\, \delta'\, \rho\, (m_1 \cup m_2)
\end{aligned}
$$

Notation 8.2.6. *We let* $\circledast_{i<n}[\, P_i \,] := P_0 * \ldots * P_{n-1}$.

Using the separation algebra laws, it is trivial to prove that our assertions form a *complete bunched implications (BI) algebra* [Pym02]. That is, they form a complete Heyting algebra in which the separating conjunction $*$ is associative, commutative, monotone, emp is neutral for $*$, and the magic wand enjoys the following introduction and elimination rules:

$$
\frac{P * Q \models_{\Gamma,\delta} R}{P \models_{\Gamma,\delta} Q \mathrel{-\!\!*} R}
\qquad\qquad
\frac{}{P * (P \mathrel{-\!\!*} Q) \models_{\Gamma,\delta} Q}
$$

In order to state properties of expressions in assertions, we define the *evaluation assertion* $e \Downarrow \nu$, which asserts that a pure expression e yields an address or value ν.

Definition 8.2.7. *The* evaluation assertion *is defined as:*

$$
\begin{aligned}
e \Downarrow \nu &:= \lambda \Gamma\, \Delta\, \delta\, \rho\, m \,.\, \exists \tau_{\mathsf{lr}} \,.\, \Gamma, \Delta, \mathsf{types}\ \rho \vdash e : \tau_{\mathsf{lr}} \ \wedge\ [\![\, e \,]\!]_{\Gamma,\rho,m} = \nu \\
e \Downarrow - &:= \exists \nu \,.\, e \Downarrow \nu
\end{aligned}
$$

The evaluator for pure expressions $[\![\, e \,]\!]_{\Gamma,\rho,m}$ *is defined in Definition 6.9.1 on page 128.*

The evaluation assertion $e \Downarrow \nu$ implicitly guarantees that the expression e is well-typed. This is required to ensure that the assertion $e \Downarrow \nu$ is closed under weakening of typing environments. By including typing judgments in assertions, some proofs also become more convenient, as details about typing are hidden.

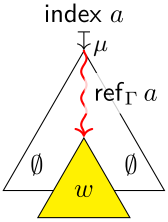
8.3 The singleton assertion

The purpose of this section is to define the *singleton assertion* and prove that it enjoys the intended properties. The shape of the singleton assertion is:

$$e_1 \overset{\gamma}{\underset{\mu}{\mapsto}} e_2 : \tau.$$

This assertion describes that the memory consists of exactly one object at address e_1 whose value e_2 has type τ and permission γ. The Boolean μ denotes whether the object has been obtained via `malloc`. As is common in separation logic, e_1 and e_2 are expressions instead of an address and value for flexibility's sake.

In order to give the definition of the singleton assertion, we will first define the *singleton memory* $\{a \mapsto w\}^{\mu}_{\Gamma}$, which is the memory that consists of exactly the memory tree w at address a. The singleton memory $\{a \mapsto w\}^{\mu}_{\Gamma}$ is depicted as follows:



This picture indicates the need for the $\emptyset$ permission. It is used to fill up the unused parts of the singleton memory with indeterminate bits $(\emptyset, \lightning)$.

Definition 8.3.1. *The* singleton memory tree $\{\vec{r} \mapsto w\}_{\Gamma}$ *is defined as:*

$$\{\varepsilon \mapsto w\}_\Gamma := w$$

$$\{(\xhookrightarrow{\tau[n]} i)\,\vec{r} \mapsto w\}_\Gamma := \mathsf{array}_\tau\ ((\mathsf{new}^\tau_\Gamma\,\emptyset)^i\ \{\vec{r} \mapsto w\}_\Gamma\ (\mathsf{new}^\tau_\Gamma\,\emptyset)^{n-i-1})$$

$$\{(\xhookrightarrow{\mathsf{struct}\ t} i)\,\vec{r} \mapsto w\}_\Gamma := \mathsf{struct}_t \begin{pmatrix} \mathsf{new}^{\tau_0}_\Gamma\,\emptyset & (\emptyset,\lightning)^{z_0} & \ldots & \mathsf{new}^{\tau_{i-1}}_\Gamma\,\emptyset & (\emptyset,\lightning)^{z_{i-1}} \\ \{\vec{r} \mapsto w\}_\Gamma & (\emptyset,\lightning)^{z_i} & & & \\ \mathsf{new}^{\tau_{i+1}}_\Gamma\,\emptyset & (\emptyset,\lightning)^{z_{i+1}} & \ldots & \mathsf{new}^{\tau_{n-1}}_\Gamma\,\emptyset & (\emptyset,\lightning)^{z_{n-1}} \end{pmatrix}$$

$$\text{where } \Gamma\,t = \vec{\tau},\ n := |\vec{\tau}|$$

$$\text{and } z_i := (\mathsf{fieldbitsizes}_\Gamma\ \vec{\tau})_i - \mathsf{bitsizeof}_\Gamma\ \tau_i$$

$$\{(\xhookrightarrow{\mathsf{union}\ t}_q i)\,\vec{r} \mapsto w\}_\Gamma := \begin{cases} \overline{\mathsf{union}}_t\ (\overline{\{\vec{r} \mapsto w\}_\Gamma}\ (\emptyset,\lightning)^z) & \text{if } \mathsf{unmapped}\ \overline{w} \\ \mathsf{union}_t\ (i, \{\vec{r} \mapsto w\}_\Gamma, (\emptyset,\lightning)^z) & \text{otherwise} \end{cases}$$

$$\text{where } \Gamma\,t = \vec{\tau} \text{ and } z := \mathsf{bitsizeof}_\Gamma\ (\mathsf{union}\ t) - \mathsf{bitsizeof}_\Gamma\ \tau_i$$

The indeterminate memory tree $\mathsf{new}^\tau_\Gamma : \mathsf{perm} \to \mathsf{mtree}$ *is defined in Definition 5.4.9 on page 76. Recall that* $\mathsf{new}^\tau_\Gamma\,\emptyset$ *consists of permission bits* $(\emptyset, \lightning)$.

Definition 8.3.2. *The* singleton memory $\{a \mapsto w\}^{\mu}_{\Gamma}$ *is defined as:*

$$\{a \mapsto w\}^{\mu}_{\Gamma} := \{(\mathsf{index}\ a,\ (\mu,\ \{\mathsf{ref}_\Gamma\ a \mapsto w\}_\Gamma))\}$$

The singleton memory $\{a \mapsto w\}^{\mu}_{\Gamma}$ is merely defined for addresses a that are strict (*i.e.* not end-of-array) and that refer to actual objects rather than individual bytes. Singletons for individual bytes are left for future work.

The expression e_2 in the singleton assertion $e_1 \overset{\gamma}{\underset{\mu}{\mapsto}} e_2 : \tau$ denotes an abstract value, which contains mathematical integers and pointers as leaves rather than a memory tree that contains bits as leaves. The encoding of abstract values as memory trees is implicit in the definition of the singleton assertion, and it therefore avoids the need to deal with bit representations explicitly. The definition is rather lengthy as it hides many details about typing and permissions.

Definition 8.3.3. *The* singleton assertion *is defined as:*

$$
\begin{aligned}
e_1 \overset{\gamma}{\underset{\mu}{\mapsto}} e_2 : \tau := \lambda \Gamma\ \Delta\ \delta\ \rho\ m\,.\,\exists a\, w\,.\, & \Gamma, \Delta, \mathsf{types}\ \rho \vdash_{\mathbf{l}} e_1 : \tau \\
& \wedge\ [\![\, e_1 \,]\!]_{\Gamma,\rho,\emptyset} = a \\
& \wedge\ \Gamma, \Delta, \mathsf{types}\ \rho \vdash_{\mathbf{r}} e_2 : \tau \\
& \wedge\ [\![\, e_2 \,]\!]_{\Gamma,\rho,\emptyset} = \mathsf{toval}_{\Gamma}\, w \\
& \wedge\ m = \{a \mapsto w\}^{\mu}_{\Gamma} \\
& \wedge\ \Gamma, \Delta \vdash w : \tau \\
& \wedge\ \text{all } \overline{w} \text{ have permission } \gamma \\
& \wedge\ a \text{ is not a byte address} \\
& \wedge\ \Gamma \vdash a\ \mathsf{strict} \\
& \wedge\ \gamma \neq \emptyset \\
e_1 \overset{\gamma}{\underset{\mu}{\mapsto}} - : \tau := \exists v\,.\, e_1 \overset{\gamma}{\underset{\mu}{\mapsto}} v : \tau &
\end{aligned}
$$

In the remainder of this section we show that the singleton assertion enjoys the intended properties that we have described in Section 4.1. First of all, we show that the singleton assertion interacts appropriately with the evaluation assertion.

Lemma 8.3.4. *If* $\mathsf{Readable} \subseteq \mathsf{kind}\ \gamma$*, then:*

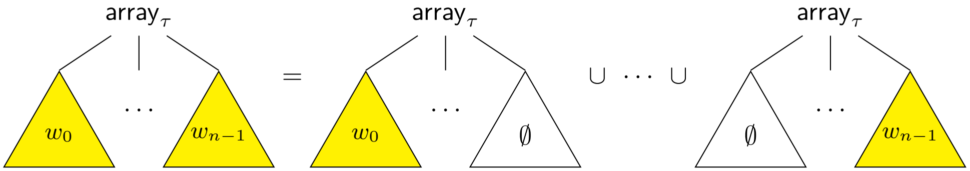
$$(e \overset{\gamma}{\underset{\mu}{\mapsto}} v : \tau) \quad \models_{\Gamma,\delta} \quad \mathsf{load}\ e \Downarrow v.$$

A singleton denoting an array value $\mathsf{array}_{\tau}\ \vec{v}$ can be subdivided into multiple singletons corresponding to each element v_i of the array.

Lemma 8.3.5 (Subdivision of arrays)**.** *If* $|\vec{v}| = n \neq 0$*, then:*

$$(e \overset{\gamma}{\underset{\mu}{\mapsto}} \mathsf{array}_{\tau}\ \vec{v} : \tau[n]) \quad \equiv_{\Gamma,\delta} \quad \circledast_{i<n} [\, (e \;._{\mathbf{l}} \xhookrightarrow{\tau[n]} i) \overset{\gamma}{\underset{\mu}{\mapsto}} v_i : \tau \,]$$

Proof. The lemma follows from the following property of singleton memory trees:



In order to prove this auxiliary property, we use the fact that the permission $\emptyset$ is a neutral element for $\cup$, from which we obtain that the indeterminate memory tree $\mathsf{new}_\Gamma^\tau\,\emptyset$ is a neutral element for $\cup$ on memory trees of type τ. □

One could furthermore prove similar results for subdivision of structs and union values, but then apart from the fields there will also be a *leftover part* that corresponds to the padding bytes of the struct or union.

The singleton assertion allows one to subdivide individual objects into multiple copies of the same object with lower permissions.

Lemma 8.3.6 (Subdivision of permissions)**.** *If* $\gamma_1 \perp \gamma_2$*, then:*

$$(e_1 \overset{\gamma_1 \cup \gamma_2}{\underset{\mu}{\longmapsto}} e_2 : \tau) \quad \equiv_{\Gamma,\delta} \quad (e_1 \overset{\gamma_1}{\underset{\mu}{\longmapsto}} e_2 : \tau) \quad * \quad (e_1 \overset{\gamma_2}{\underset{\mu}{\longmapsto}} e_2 : \tau)$$

provided $\neg\mathsf{unmapped}\ \gamma_1$ *and* $\neg\mathsf{unmapped}\ \gamma_2$*.*

Proof. The lemma follows from the following property of singleton memory trees:

$$\{\vec{r} \mapsto w_1 \cup w_2\}_\Gamma = \{\vec{r} \mapsto w_1\}_\Gamma \cup \{\vec{r} \mapsto w_2\}_\Gamma$$

□

As a corollary of this lemma we have that a writable singleton can be subdivided into read only parts. Given a permission γ with $\mathsf{Readable} \subseteq \mathsf{kind}\ \gamma$ we have:

$$\underbrace{(e_1 \overset{\gamma}{\underset{\mu}{\longmapsto}} e_2 : \tau)}_{\text{Writable or read only}} \quad \equiv_{\Gamma,\delta} \quad \underbrace{(e_1 \overset{\frac{1}{2}\gamma}{\underset{\mu}{\longmapsto}} e_2 : \tau)}_{\text{Read only}} \quad * \quad \underbrace{(e_1 \overset{\frac{1}{2}\gamma}{\underset{\mu}{\longmapsto}} e_2 : \tau)}_{\text{Read only}}$$

The situation is slightly different in case of unmapped permissions because then the contents of the object is not subdivided among both parts. In particular, if we wish to subdivide a writable singleton into an existing and writable part we should have the following for each γ with $\mathsf{Readable} \subseteq \mathsf{kind}\ \gamma$:

$$\underbrace{(e_1 \overset{\gamma}{\underset{\mu}{\longmapsto}} e_2 : \tau)}_{\text{Writable or read only}} \quad \equiv_{\Gamma,\delta} \quad \underbrace{(e_1 \overset{\mathsf{token}}{\underset{\mu}{\longmapsto}} - : \tau)}_{\text{Existing}} \quad * \quad \underbrace{(e_1 \overset{\gamma \setminus \mathsf{token}}{\underset{\mu}{\longmapsto}} e_2 : \tau)}_{\text{Writable or read only}}$$

Note that the existing part has an arbitrary value. The above property is a trivial corollary of the following lemma.

Lemma 8.3.7 (Subdivision of unmapped permissions)**.** *If* $\gamma_1 \perp \gamma_2$*, then:*

$$(e_1 \overset{\gamma_1 \cup \gamma_2}{\underset{\mu}{\longmapsto}} e_2 : \tau) \quad \equiv_{\Gamma,\delta} \quad (e_1 \overset{\gamma_1}{\underset{\mu}{\longmapsto}} - : \tau) \quad * \quad (e_1 \overset{\gamma_2}{\underset{\mu}{\longmapsto}} e_2 : \tau)$$

provided that $\gamma_1 \neq \emptyset$ *and* $\neg\mathsf{unmapped}\ \gamma_2$*.*

This lemma is the cause of the restrictions on unions that we have introduced in the separation algebra relation $\perp$ and operation $\cup$ on memory trees (Definition 8.1.4). Without these restrictions on unions, the above lemma would not hold.

These restrictions ensure that a writable union could only be subdivided into an existing and writable part as follows:

$$\begin{array}{ccccc} \mathsf{union}_t & & \overline{\mathsf{union}_t} & & \mathsf{union}_t \\ \big|\,.i & = & \big| & \cup & \big|\,.i \\ \text{Writable part} & & \text{Existing part} & & \text{Writable part} \end{array}$$

As shown, the existing part has an unspecified variant rather than variant i. The reason for doing so is that the variant of the writable part may change at run-time in case the union is accessed via a pointer to another variant. We thus require unions with unmapped permissions to have an unspecified variant. Unions with unspecified variant are neutral elements for $\cup$ with respect to any other union.

8.4 Separation logic for expressions

This section defines the judgment $\vdash_{\Gamma,\delta} \{P\}\, e\, \{Q\}$ for expressions. Like Von Oheimb [Ohe01], we let the postcondition Q be a function from values to assertions to account for the fact that an expression not only performs side-effects but primarily yields a value. The judgment $\vdash_{\Gamma,\delta} \{P\}\, e\, \{Q\}$ should be read as follows: if P holds for the memory beforehand, and execution of e yields an address or value ν, then $Q\,\nu$ holds for the resulting memory afterwards.

The key observation that led to the inference rules of our judgment for expressions is a correspondence between non-determinism in expressions and concurrency. Inspired by the rule for the parallel composition of concurrent separation logic [O'H04], we have rules for each operator $\circledcirc$ that are of the following shape:

$$\frac{\{P_1\}\, e_1\, \{Q_1\} \qquad \{P_2\}\, e_2\, \{Q_2\}}{\{P_1 * P_2\}\, e_1 \circledcirc e_2\, \{Q_1 * Q_2\}}$$

The intuitive idea of this rule is that if the memory can be subdivided into two parts in which the subexpressions e_1 and e_2 can be executed safely, then the expression $e_1 \circledcirc e_2$ can be executed safely in the whole memory. Non-interference of the side-effects of e_1 and e_2 is guaranteed by the separating conjunction. We thus effectively rule out expressions with undefined behavior due to a sequence point violation such as `(x = 3) + (x = 4)` (see Sections 2.5.9 and 4.1 for discussion).

The actual rules of our expression judgment are more complicated than the rule sketched above because we have to deal with the values of expressions and we have to make sure that no undefined behavior due to for example integer overflow occurs. Before considering the actual rules in Definition 8.4.4, we discuss some representative rules. The actual rule for binary operators is:

$$\frac{\begin{array}{c}\vdash_{\Gamma,\delta} \{P_1\}\, e_1\, \{Q_1\} \qquad \vdash_{\Gamma,\delta} \{P_2\}\, e_2\, \{Q_2\} \\ \forall v_1\, v_2\, .\, (Q_1\, v_1 * Q_2\, v_2 \models_{\Gamma,\delta} \exists v'\, .\, (v_1 \circledcirc v_2) \Downarrow v' \land Q'\, v')\end{array}}{\vdash_{\Gamma,\delta} \{P_1 * P_2\}\, e_1 \circledcirc e_2\, \{Q'\}}$$

The side-condition ensures that for each value v_1 of e_1 and value v_2 of e_2, the binary operator can be evaluated safely to a resulting value v' for which the postcondition

$Q'\,v'$ of the whole expression holds. To express that a binary operator can be evaluated safely we use the evaluation assertion $(v_1 \circledcirc v_2) \Downarrow v'$ (Definition 8.2.7).

In order to ensure the absence of sequence point violations, the rule for assignments changes the permission γ of the assigned object into $\mathsf{lock}\ \gamma$. Subsequent accesses are therefore no longer possible. The rule for the simple assignments is:

$$\frac{\begin{array}{c}\vdash_{\Gamma,\delta} \{P_1\}\, e_1\, \{Q_1\} \qquad \vdash_{\Gamma,\delta} \{P_2\}\, e_2\, \{Q_2\} \qquad \mathsf{Writable} \subseteq \mathsf{kind}\ \gamma \\ \forall p\, v\,.\, \Big(Q_1\, p * Q_2\, v \models_{\Gamma,\delta} \exists v'\,.\,(\tau)v \Downarrow v' \wedge \\ \qquad\qquad ((p \xmapsto[\mu]{\gamma} - : \tau) * ((p \xmapsto[\mu]{\mathsf{lock}\ \gamma} |\, v'\,|_\circ : \tau) \mathrel{-\!\!*} Q'\, v'))\Big)\end{array}}{\vdash_{\Gamma,\delta} \{P_1 * P_2\}\, e_1 := e_2\, \{Q'\}}$$

The assertion $(p \xmapsto[\mu]{\gamma} - : \tau) * ((p \xmapsto[\mu]{\mathsf{lock}\ \gamma} |\, v'\,|_\circ : \tau) \mathrel{-\!\!*} Q'\, v')$ ensures that each pointer p obtained from e_1 is writable. The magic wand is then used to replace the value of p by $|\, v'\,|_\circ$ and to lock the permission. The stored value $|\, v'\,|_\circ$ is frozen.

At constructs that have a sequence point, we have to release the locks of objects that have been locked due to previous assignments. We define the assertion $P\,\Diamond$ that releases all locks in P and thereby makes future reads and writes possible again. The rule for the comma expression is as follows:

$$\frac{\vdash_{\Gamma,\delta} \{P\}\, e_1\, \{\lambda\_\,.\, P'\,\Diamond\} \qquad \vdash_{\Gamma,\delta} \{P'\}\, e_2\, \{Q\}}{\vdash_{\Gamma,\delta} \{P\}\, (e_1, e_2)\, \{Q\}}$$

Definition 8.4.1. *The* unlocking assertion $P\,\Diamond$ *is defined as:*

$$P\,\Diamond := \lambda \Gamma\, \Delta\, \delta\, \rho\, m\,.\, P\, \Gamma\, \Delta\, \delta\, \rho\, (\mathsf{unlock}\ (\mathsf{locks}\ m)\ m).$$

Lemma 8.4.2. *For each permission γ we have:*

$$\begin{array}{rl} (e_1 \xmapsto[\mu]{\mathsf{lock}\ \gamma} e_2 : \tau_{\mathsf{lr}}) \models_{\Gamma,\delta} (e_1 \xmapsto[\mu]{\gamma} e_2 : \tau_{\mathsf{lr}})\,\Diamond & \qquad \text{if } \mathsf{Writable} \subseteq \mathsf{kind}\ \gamma \\ (e_1 \xmapsto[\mu]{\gamma} e_2 : \tau_{\mathsf{lr}}) \models_{\Gamma,\delta} (e_1 \xmapsto[\mu]{\gamma} e_2 : \tau_{\mathsf{lr}})\,\Diamond & \qquad \text{if } \mathsf{kind}\ \gamma \neq \mathsf{Locked} \end{array}$$

Moreover, $\Diamond$ distributes over all logical connectives:

$$\begin{array}{lll} P\,\Diamond \to Q\,\Diamond \models_{\Gamma,\delta} (P \to Q)\,\Diamond & \forall x\,.\,(P\,x)\,\Diamond \models_{\Gamma,\delta} (\forall x\,.\,P\,x)\,\Diamond & \ulcorner A \urcorner \models_{\Gamma,\delta} \ulcorner A \urcorner\,\Diamond \\ P\,\Diamond \wedge Q\,\Diamond \models_{\Gamma,\delta} (P \wedge Q)\,\Diamond & \exists x\,.\,(P\,x)\,\Diamond \models_{\Gamma,\delta} (\exists x\,.\,P\,x)\,\Diamond & P\,\Diamond * Q\,\Diamond \models_{\Gamma,\delta} (P * Q)\,\Diamond \end{array}$$

In order to define the assignment rule in its full generality, we have to deal with all forms of assignments. The following definition describes an assertion that corresponds to Definition 6.3.7 on page 104, which is its counterpart in the operational semantics.

Definition 8.4.3. *The assertion $(p\ \alpha\ v : \tau) \Downarrow_{\mathsf{ass}} (v_a,\, v')$ describes the assigned value v_a and resulting r-value v' of an assignment $a\ \alpha\ v$ of type τ. It is defined as:*

$$\frac{(\tau)v \Downarrow v_a \qquad (\tau)v \Downarrow v'}{(p := v : \tau) \Downarrow_{\mathsf{ass}} (v_a,\, v')} \qquad \frac{(\tau)(\mathsf{load}\ a \circledcirc v) \Downarrow v_a \qquad (\tau)(\mathsf{load}\ a \circledcirc v) \Downarrow v'}{(p \circledcirc\!:= v : \tau) \Downarrow_{\mathsf{ass}} (v_a,\, v')}$$

$$\frac{(\tau)(\mathsf{load}\ a \circledcirc v) \Downarrow v_a \qquad \mathsf{load}\ a \Downarrow v'}{(p :=\!\circledcirc\, v : \tau) \Downarrow_{\mathsf{ass}} (v_a,\, v')}$$

We now give the full set of the rules of the expression judgments. These cover all expression constructs apart from function calls, which are discussed in Section 8.7. The rules of most expression constructs are similar to those we have demonstrated above. We will comment on some specific rules after the definition.

In the Coq formalization we have defined the judgment using a shallow embedding. The inference rules are there just lemmas.

Definition 8.4.4. *The judgment* $\vdash_{\Gamma,\delta} \{P\}\, e\, \{Q\}$ *of the* axiomatic semantics for expressions *is inductively defined as:*

$$\frac{\vdash_{\Gamma,\delta} \{P\}\, e\, \{Q\}}{\vdash_{\Gamma,\delta} \{A * P\}\, e\, \{A * Q\}} \qquad \frac{\forall x \,.\, (\vdash_{\Gamma,\delta} \{P\, x\}\, e\, \{Q\})}{\vdash_{\Gamma,\delta} \{\exists x \,.\, P\, x\}\, e\, \{Q\}}$$

$$\frac{P' \models_{\Gamma,\delta} P \qquad \vdash_{\Gamma,\delta} \{P\}\, e\, \{Q\} \qquad \forall \nu \,.\, (Q\, \nu \models_{\Gamma,\delta} Q'\, \nu)}{\vdash_{\Gamma,\delta} \{P'\}\, e\, \{Q'\}} \qquad \frac{P \models_{\Gamma,\delta} e \Downarrow \nu \wedge Q\, \nu}{\vdash_{\Gamma,\delta} \{P\}\, e\, \{Q\}}$$

$$\frac{\begin{array}{c}\vdash_{\Gamma,\delta} \{P\}\, e\, \{Q\} \\ \forall v \,.\, (Q\, v \models_{\Gamma,\delta} \exists p' \,.\, ({*}v) \Downarrow p' \wedge Q'\, p')\end{array}}{\vdash_{\Gamma,\delta} \{P\}\, {*}e\, \{Q'\}} \qquad \frac{\begin{array}{c}\vdash_{\Gamma,\delta} \{P\}\, e\, \{Q\} \\ \forall p \,.\, (Q\, p \models_{\Gamma,\delta} \exists v' \,.\, (\&p) \Downarrow v' \wedge Q'\, v')\end{array}}{\vdash_{\Gamma,\delta} \{P\}\, \&e\, \{Q'\}}$$

$$\frac{\begin{array}{c}\vdash_{\Gamma,\delta} \{P\}\, e\, \{Q\} \\ \forall p \,.\, (Q\, p \models_{\Gamma,\delta} \exists p' \,.\, (p \mathbin{._{\mathbf{l}}} r) \Downarrow p' \wedge Q'\, p')\end{array}}{\vdash_{\Gamma,\delta} \{P\}\, e \mathbin{._{\mathbf{l}}} r\, \{Q'\}} \qquad \frac{\begin{array}{c}\vdash_{\Gamma,\delta} \{P\}\, e\, \{Q\} \\ \forall v \,.\, (Q\, v \models_{\Gamma,\delta} \exists v' \,.\, (v \mathbin{._{\mathbf{r}}} r) \Downarrow v' \wedge Q'\, v')\end{array}}{\vdash_{\Gamma,\delta} \{P\}\, e \mathbin{._{\mathbf{r}}} r\, \{Q'\}}$$

$$\frac{\begin{array}{c}\vdash_{\Gamma,\delta} \{P_1\}\, e_1\, \{Q_1\} \qquad \vdash_{\Gamma,\delta} \{P_2\}\, e_2\, \{Q_2\} \\ \forall v_1\, v_2 \,.\, (Q_1\, v_1 * Q_2\, v_2 \models_{\Gamma,\delta} \exists v' \,.\, (v_1[\vec{r} := v_2]) \Downarrow v' \wedge Q'\, v')\end{array}}{\vdash_{\Gamma,\delta} \{P_1 * P_2\}\, e_1[\vec{r} := e_2]\, \{Q'\}}$$

$$\frac{\begin{array}{c}\vdash_{\Gamma,\delta} \{P_1\}\, e_1\, \{Q_1\} \qquad \vdash_{\Gamma,\delta} \{P_2\}\, e_2\, \{Q_2\} \qquad \mathsf{Writable} \subseteq \mathsf{kind}\ \gamma \\ \forall p\, v \,.\, \Big(Q_1\, p * Q_2\, v \models_{\Gamma,\delta} \exists v_a\, v' \,.\, (p\ \alpha\ v : \tau) \Downarrow_{\mathsf{ass}} (v_a,\, v') \wedge \\ \qquad\qquad ((p \xmapsto[\mu]{\gamma} - : \tau) * ((p \xmapsto[\mu]{\mathsf{lock}\ \gamma} |\, v_a\,|_{\circ} : \tau) \mathrel{-\!\!*} Q'\, v'))\Big)\end{array}}{\vdash_{\Gamma,\delta} \{P_1 * P_2\}\, e_1\ \alpha\ e_2\, \{Q'\}}$$

$$\frac{\vdash_{\Gamma,\delta} \{P\}\, e\, \{Q\} \qquad \forall p \,.\, (Q\, p \models_{\Gamma,\delta} \exists v' \,.\, (\mathsf{load}\ p) \Downarrow v' \wedge Q'\, v')}{\vdash_{\Gamma,\delta} \{P\}\, \mathsf{load}\ e\, \{Q'\}}$$

$$\frac{\begin{array}{c}\vdash_{\Gamma,\delta} \{P\}\, e\, \{Q\} \\ \forall o\, v_{\mathsf{b}} \,.\, \Big(Q\, v_{\mathsf{b}} \models_{\Gamma,\delta} \exists n\, \tau_{\mathsf{i}} \,.\, \ulcorner v_{\mathsf{b}} = \mathsf{int}_{\tau_{\mathsf{i}}}\, n \urcorner * \ulcorner 0 < n \urcorner * \\ \qquad\qquad ((\mathsf{top}_{\tau[n]}\, o \xmapsto[\mathsf{true}]{\Diamond(0,1)} - : \tau[n]) \mathrel{-\!\!*} Q'\, ((\mathsf{top}_{\tau}\, o)\langle \xhookrightarrow{\tau[n]} 0 \rangle_{\Gamma}))\Big)\end{array}}{\vdash_{\Gamma,\delta} \{P\}\, \mathsf{alloc}_{\tau}\ e\, \{Q'\}}$$

$$\frac{\vdash_{\Gamma,\delta} \{P\}\, e\, \{Q\} \qquad \begin{array}{r} \forall p\,.\,\big(Q\, a \models_{\Gamma,\delta} \exists n\; o\; \tau_{\mathsf{p}}\,.\,\ulcorner p = (o : \tau[n], \xhookrightarrow{\tau[n]} 0, 0)_{\tau >_{*} \tau_{\mathsf{p}}} \urcorner * \\ (\mathsf{top}_{\tau[n]}\; o \overset{\Diamond(0,1)}{\underset{\mathsf{true}}{\longmapsto}} - : \tau[n]) * Q'\, \mathsf{nothing}\big) \end{array}}{\vdash_{\Gamma,\delta} \{P\}\, \mathsf{free}\; e\, \{Q'\}}$$

$$\frac{\begin{array}{c} \vdash_{\Gamma,\delta} \{P_1\}\, e_1\, \{Q_1\} \qquad \vdash_{\Gamma,\delta} \{P_2\}\, e_2\, \{Q_2\} \\ \forall v_1\, v_2\,.\,(Q_1\, v_1 * Q_2\, v_2 \models_{\Gamma,\delta} \exists v'\,.\,(v_1 \circledcirc v_2) \Downarrow v' \wedge Q'\, v') \end{array}}{\vdash_{\Gamma,\delta} \{P_1 * P_2\}\, e_1 \circledcirc e_2\, \{Q'\}}$$

$$\frac{\begin{array}{c} \vdash_{\Gamma,\delta} \{P\}\, e\, \{Q\} \\ \forall v\,.\,(Q\, v \models_{\Gamma,\delta} \exists v'\,.\,(\circledcirc_u\, v) \Downarrow v' \wedge Q'\, v') \end{array}}{\vdash_{\Gamma,\delta} \{P\}\, \circledcirc_u\; e\, \{Q'\}} \qquad \frac{\begin{array}{c} \vdash_{\Gamma,\delta} \{P\}\, e\, \{Q\} \\ \forall v\,.\,(Q\, v \models_{\Gamma,\delta} \exists v'\,.\,((\tau)v) \Downarrow v' \wedge Q'\, v') \end{array}}{\vdash_{\Gamma,\delta} \{P\}\, (\tau)e\, \{Q'\}}$$

$$\frac{\begin{array}{cc} \vdash_{\Gamma,\delta} \{P\}\, e\, \{P'\} & \forall v_{\mathsf{b}}\,.\,(P'\, v_{\mathsf{b}} \models_{\Gamma,\delta} (!\; v_{\mathsf{b}}) \Downarrow -) \\ \vdash_{\Gamma,\delta} \{P_1\}\, e_1\, \{Q\} & \forall v_{\mathsf{b}}\,.\,(\neg \mathsf{zero}\; v_{\mathsf{b}} \to P'\, v_{\mathsf{b}} \models_{\Gamma,\delta} P_1\, \Diamond) \\ \vdash_{\Gamma,\delta} \{P_2\}\, e_2\, \{Q\} & \forall v_{\mathsf{b}}\,.\,(\mathsf{zero}\; v_{\mathsf{b}} \to P'\, v_{\mathsf{b}} \models_{\Gamma,\delta} P_2\, \Diamond) \end{array}}{\vdash_{\Gamma,\delta} \{P\}\, e \;?\; e_1 : e_2\, \{Q\}} \qquad \frac{\begin{array}{c} \vdash_{\Gamma,\delta} \{P\}\, e_1\, \{\lambda\_\,.\, P'\, \Diamond\} \\ \vdash_{\Gamma,\delta} \{P'\}\, e_2\, \{Q\} \end{array}}{\vdash_{\Gamma,\delta} \{P\}\, (e_1, e_2)\, \{Q\}}$$

The expression judgment has a frame, weaken, and exist rule. The weaken and exist rule are similar to their traditional counterparts in Hoare logic. The conventional frame rule of separation logic [ORY01] includes a side-condition $\mathsf{modifies}\, e \cap \mathsf{free}\, A = \emptyset$ on the free variables of the expression e and assertion A. We do not need such a side-condition as our local variables are (immutable) references into the memory.

The rules for $\mathsf{alloc}_\tau\; e$ and $\mathsf{free}\; e$ constructs, which are used to allocate and deallocate dynamically obtained memory, are somewhat complicated. This complication is caused by the fact that $\mathsf{alloc}_\tau\; e$ yields a pointer to the first element of the allocated array whereas the singleton in the postcondition ranges over the entire array. In the Coq development we have defined some auxiliary notions to make these rules more tractable, but for clarity's sake, we have expanded these notions here. For simplicity we presuppose that $\mathsf{alloc}_\tau\; e$ will never return a `NULL` pointer in this thesis[2].

The rule for the conditional $e \;?\; e_1 : e_2$ contains the condition $P'\, v_{\mathsf{b}} \models_{\Gamma,\delta} (!\; v_{\mathsf{b}}) \Downarrow -$, which ensures that the result v_{b} of e is not indeterminate. We use that the condition $m \vdash (!\; v_{\mathsf{b}})\; \mathsf{defined}$ of the negation operator (Definition 6.3.5 on page 104) is the same as the one $m \vdash (\mathsf{zero}\; v_{\mathsf{b}})\; \mathsf{defined}$ of the conditional (Definition 6.3.6 on page 104).

8.5 Separation logic for statements

This section defines the judgment $R, J, T \vdash_{\Gamma,\delta} \{P\}\, s\, \{Q\}$ for statements. The right hand side of the judgment is a conventional Hoare triple. That is, if P holds for the memory beforehand, and execution of s terminates, then Q holds for the resulting memory afterwards. The judgment furthermore guarantees the absence of undefined behavior. The environments R, J and T deal with non-local control:

[2] The implementation environments in our Coq development have a flag `alloc_can_fail`, which describes if $\mathsf{alloc}_\tau\; e$ may fail. If set, the $\mathsf{alloc}_\tau\; e$ rule contains a conjunct to account for failure.

- The environment $R : \mathsf{val} \to \mathsf{assert}$ specifies the conditions for **`return`** statements. The assertion $R\,v$ should hold when executing a $\mathsf{return}\ e$ for each value v that can be obtained by executing e.
- The environment $J : \mathsf{labelname} \to \mathsf{assert}$ specifies the conditions for **`goto`**s. The assertion $J\,l$ is the jumping condition that should hold when executing a $\mathsf{goto}\ l$ statement.
- The environment $T : \mathbb{N} \to \mathsf{assert}$ specifies the conditions for `throw` statements (which generalize **`break`** and **`continue`**). The assertion $T\,n$ is the jumping condition that should hold when executing a $\mathsf{throw}\ n$ statement.

Our treatment of **`return`**, `throw` and `catch` statements is adapted from Appel and Blazy [AB07]. Let us consider the rules for `throw` and `catch`:

$$\frac{}{R, J, T \vdash_{\Gamma,\delta} \{T\ n\}\,\mathsf{throw}\ n\,\{Q\}} \qquad \frac{R, J, Q \cdot T \vdash_{\Gamma,\delta} \{P\}\,s\,\{Q\}}{R, J, T \vdash_{\Gamma,\delta} \{P\}\,\mathsf{catch}\ s\,\{Q\}}$$

As shown, the environment T is used to relate the precondition of each `throw` to the postcondition of its corresponding `catch`. The assertion $(Q \cdot T) : \mathbb{N} \to \mathsf{assert}$ is defined as $(Q \cdot T)\,0 := Q$ and $(Q \cdot T)\,(n+1) := T\,n$. Since the `throw` statement leaves the normal control flow, its postcondition is arbitrary. We have extended Appel and Blazy's treatment with **`goto`**s in the expected way:

$$\frac{}{R, J, T \vdash_{\Gamma,\delta} \{J\ l\}\,\mathsf{goto}\ l\,\{Q\}} \qquad \frac{}{R, J, T \vdash_{\Gamma,\delta} \{J\ l\}\,l :\ \{J\ l\}}$$

Non-local control flow becomes more interesting in the presence of block scope local variables. Jumping into a block scope results in allocation of a new local variable, to which pointers can be created. Likewise, when jumping out of a block scope, the corresponding local variable will be deallocated, and pointers to it become indeterminate. This symmetry becomes clear in the following rule:

$$\frac{\begin{array}{r}(x_0 \xmapsto[\mathsf{false}]{\Diamond(0,1)} - : \tau) * R\uparrow,\ (x_0 \xmapsto[\mathsf{false}]{\Diamond(0,1)} - : \tau) * J\uparrow,\ (x_0 \xmapsto[\mathsf{false}]{\Diamond(0,1)} - : \tau) * T\uparrow \\ \vdash_{\Gamma,\delta} \{(x_0 \xmapsto[\mathsf{false}]{\Diamond(0,1)} - : \tau) * P\uparrow\}\,s\,\{(x_0 \xmapsto[\mathsf{false}]{\Diamond(0,1)} - : \tau) * Q\uparrow\}\end{array}}{R, J, T \vdash_{\Gamma,\delta} \{P\}\,\mathsf{local}_\tau\ s\,\{Q\}}$$

When entering a block scope the De Bruijn indexes are lifted using the assertion $(\_)\uparrow$ (defined below) and in turn the memory is extended in a way that the variable x_0 refers to the newly allocated object. Allocation is described using the separating conjunction. When leaving a block, the inverse takes place. This symmetry is important for **`goto`**s, which can jump into block scopes as well as out of block scopes. Note that the use of De Bruijn indexes avoids having to deal with shadowing.

In the above rule for block scopes, we need to lift an assertion such that the De Bruijn indexes of its variables are increased. We define the lifting $P\uparrow$ of an assertion P semantically, and prove that it indeed behaves as expected.

Definition 8.5.1. *The* lifting assertion $P\uparrow$ *is defined as:*

$$P\uparrow := \lambda\,\Gamma\,\Delta\,\delta\,\rho\,m\,.\,P\,\Gamma\,\Delta\,\delta\,(\mathsf{tail}\,\rho)\,m.$$

Lemma 8.5.2. *The operation* $(\_)\uparrow$ *distributes over the defined assertions:*

$$\begin{array}{ccc} (P \to Q)\uparrow \equiv_{\Gamma,\delta} P\uparrow \to Q\uparrow & (\forall x\,.\,P\,x)\uparrow \equiv_{\Gamma,\delta} \forall x\,.\,(P\,x)\uparrow & \ulcorner A \urcorner\uparrow \equiv_{\Gamma,\delta} \ulcorner A \urcorner \\ (P \wedge Q)\uparrow \equiv_{\Gamma,\delta} P\uparrow \wedge Q\uparrow & (\exists x\,.\,P\,x)\uparrow \equiv_{\Gamma,\delta} \exists x\,.\,(P\,x)\uparrow & (P * Q)\uparrow \equiv_{\Gamma,\delta} P\uparrow * Q\uparrow \end{array}$$

Apart from that, we have:

$$(e \Downarrow \nu)\uparrow \equiv_{\Gamma,\delta} (e\uparrow) \Downarrow \nu \qquad\qquad (e_1 \xmapsto[\mu]{\gamma} e_2 : \tau)\uparrow \equiv_{\Gamma,\delta} (e_1\uparrow) \xmapsto[\mu]{\gamma} (e_2\uparrow) : \tau$$

where the expression $e\uparrow$ *is obtained by replacing each variable* x_i *by* x_{i+1}.

Definition 8.5.3. *The judgment* $R, J, T \vdash_{\Gamma,\delta} \{P\}\,s\,\{Q\}$ *of the* axiomatic semantics for statements *is inductively defined as:*

$$\frac{R, J, T \vdash_{\Gamma,\delta} \{P\}\,s\,\{Q\}}{A * R, A * J, A * T \vdash_{\Gamma,\delta} \{A * P\}\,s\,\{A * Q\}} \qquad \frac{\forall x\,.\,(R, J, T \vdash_{\Gamma,\delta} \{P\,x\}\,s\,\{Q\})}{R, J, T \vdash_{\Gamma,\delta} \{\exists x\,.\,P\,x\}\,s\,\{Q\}}$$

$$\frac{\begin{array}{c} \forall v\,.\,(R\,v \models_{\Gamma,\delta} R'\,v) \qquad \forall l \in \mathsf{labels}\ s\,.\,(J'\,l \models_{\Gamma,\delta} J\,l) \qquad \forall l \notin \mathsf{labels}\ s\,.\,(J\,l \models_{\Gamma,\delta} J'\,l) \\ \forall n\,.\,(T\,n \models_{\Gamma,\delta} T'\,n) \qquad P' \models_{\Gamma,\delta} P \qquad R, J, T \vdash_{\Gamma,\delta} \{P\}\,s\,\{Q\} \qquad Q \models_{\Gamma,\delta} Q' \end{array}}{R', J', T' \vdash_{\Gamma,\delta} \{P'\}\,s\,\{Q'\}}$$

$$\frac{\vdash_{\Gamma,\delta} \{P\}\,e\,\{\lambda\_\,.\,Q\,\Diamond\}}{R, J, T \vdash_{\Gamma,\delta} \{P\}\,e\,\{Q\}} \qquad \frac{\vdash_{\Gamma,\delta} \{P\}\,e\,\{Q\} \qquad \forall v\,.\,(Q\,v \models_{\Gamma,\delta} (R\,v)\,\Diamond)}{R, J, T \vdash_{\Gamma,\delta} \{P\}\,\mathsf{return}\ e\,\{Q'\}}$$

$$\frac{}{R, J, T \vdash_{\Gamma,\delta} \{J\,l\}\,\mathsf{goto}\ l\,\{Q\}} \qquad \frac{}{R, J, T \vdash_{\Gamma,\delta} \{J\,l\}\,l:\ \{J\,l\}}$$

$$\frac{}{R, J, T \vdash_{\Gamma,\delta} \{T\,n\}\,\mathsf{throw}\ n\,\{Q\}} \qquad \frac{R, J, Q \cdot T \vdash_{\Gamma,\delta} \{P\}\,s\,\{Q\}}{R, J, T \vdash_{\Gamma,\delta} \{P\}\,\mathsf{catch}\ s\,\{Q\}}$$

$$\frac{}{R, J, T \vdash_{\Gamma,\delta} \{P\}\,\mathsf{skip}\,\{P\}} \qquad \frac{R, J, T \vdash_{\Gamma,\delta} \{P\}\,s_1\,\{P'\} \qquad R, J, T \vdash_{\Gamma,\delta} \{P'\}\,s_2\,\{Q\}}{R, J, T \vdash_{\Gamma,\delta} \{P\}\,s_1\,;\ s_2\,\{Q\}}$$

$$\frac{\begin{array}{l} (x_0 \xmapsto[\mathsf{false}]{\Diamond(0,1)} - : \tau) * R\uparrow,\ (x_0 \xmapsto[\mathsf{false}]{\Diamond(0,1)} - : \tau) * J\uparrow,\ (x_0 \xmapsto[\mathsf{false}]{\Diamond(0,1)} - : \tau) * T\uparrow \\ \qquad\qquad \vdash_{\Gamma,\delta} \{(x_0 \xmapsto[\mathsf{false}]{\Diamond(0,1)} - : \tau) * P\uparrow\}\,s\,\{(x_0 \xmapsto[\mathsf{false}]{\Diamond(0,1)} - : \tau) * Q\uparrow\} \end{array}}{R, J, T \vdash_{\Gamma,\delta} \{P\}\,\mathsf{local}_\tau\ s\,\{Q\}}$$

$$\frac{R, J, T \vdash_{\Gamma,\delta} \{P\}\,s\,\{P\}}{R, J, T \vdash_{\Gamma,\delta} \{P\}\,\mathsf{loop}\ s\,\{\mathsf{False}\}}$$

$$\frac{\begin{array}{rl} \vdash_{\Gamma,\delta} \{P\}\,e\,\{P'\} & \forall v_{\mathsf{b}}\,.\,(P'\,v_{\mathsf{b}} \models_{\Gamma,\delta} (!\ v_{\mathsf{b}}) \Downarrow -) \\ R, J, T \vdash_{\Gamma,\delta} \{P_1\}\,s_1\,\{Q\} & \forall v_{\mathsf{b}}\,.\,(\neg \mathsf{zero}\ v_{\mathsf{b}} \to P'\,v_{\mathsf{b}} \models_{\Gamma,\delta} P_1\,\Diamond) \\ R, J, T \vdash_{\Gamma,\delta} \{P_2\}\,s_2\,\{Q\} & \forall v_{\mathsf{b}}\,.\,(\mathsf{zero}\ v_{\mathsf{b}} \to P'\,v_{\mathsf{b}} \models_{\Gamma,\delta} P_2\,\Diamond) \end{array}}{R, J, T \vdash_{\Gamma,\delta} \{P\}\,\mathsf{if}\ (e)\ s_1\ \mathsf{else}\ s_2\,\{Q\}}$$

The assertions P, Q, J, R and T correspond to the directions $\searrow$, $\nearrow$, $\curvearrowright$, $\upuparrows$ and $\uparrow$ of the operational semantics in which traversal through a statement is performed (see Definition 6.5.5 on page 116). We therefore introduce a shorthand.

Notation 8.5.4. *The triple $R, J, T \vdash_{\Gamma,\delta} \{P\}\, s\, \{Q\}$ is sometimes denoted as $\bar{P} \vdash_{\Gamma,\delta} s$, where $\bar{P} : \mathsf{direction} \to \mathsf{assert}$ is defined as:*

$$\bar{P}\searrow := P \qquad \bar{P}\nearrow := Q \qquad \bar{P}\,(\curvearrowright l) := J\,l \qquad \bar{P}\,(\upuparrows v) := R\,v \qquad \bar{P}\,(\uparrow n) := T\,n.$$

Being able to write sextuples in a shorter way is convenient while proving metatheoretical results, such as soundness of the axiomatic semantics in Section 8.6. The frame rule and the rule for block scope local variables can be abbreviated since these are uniform in all assertions:

$$\frac{\bar{P} \vdash_{\Gamma,\delta} s}{A * \bar{P} \vdash_{\Gamma,\delta} s} \qquad \frac{(x_0 \xmapsto[\mathsf{false}]{\Diamond(0,\,1)} - : \tau) * \bar{P}\uparrow \vdash_{\Gamma,\delta} s}{\bar{P} \vdash_{\Gamma,\delta} \mathsf{local}_\tau\, s}$$

8.6 Soundness of the separation logic

This section defines the judgments $\models_{\Gamma,\delta} \{P\}\, e\, \{Q\}$ and $J, R, T \models_{\Gamma,\delta} \{P\}\, s\, \{Q\}$. These judgments describe partial program correctness in terms of the CH$_2$O core C semantics. Soundness of the separation logic means that (Theorem 8.6.13):

$$\begin{array}{rcr} \vdash_{\Gamma,\delta} \{P\}\, e\, \{Q\} & \text{implies} & \models_{\Gamma,\delta} \{P\}\, e\, \{Q\} \\ J, R, T \vdash_{\Gamma,\delta} \{P\}\, s\, \{Q\} & \text{implies} & J, R, T \models_{\Gamma,\delta} \{P\}\, s\, \{Q\} \end{array}$$

In the Coq development we have used a shallow embedding, and therefore did not define the judgments $\vdash_{\Gamma,\delta} \{P\}\, e\, \{Q\}$ and $J, R, T \vdash_{\Gamma,\delta} \{P\}\, s\, \{Q\}$ explicitly. The inference rules that we have defined in Definitions 8.4.4 and 8.5.3 are there thus just lemmas involving the shallow embedding.

The definition of the judgments $\models_{\Gamma,\delta} \{P\}\, e\, \{Q\}$ and $J, R, T \models_{\Gamma,\delta} \{P\}\, s\, \{Q\}$ is not so simple. Before we will consider the actual definitions, we consider a naive attempt to define the judgment $\models_{\Gamma,\delta} \{P\}\, e\, \{Q\}$:

Attempt 8.6.1. *For each well-typed context $\mathcal{P}$ and memory m_1 in which the precondition $P\ \Gamma\ \overline{m_1}\ \delta\ (\mathsf{locals}\ \mathcal{P})\ m_1$ holds, and for each reduction:*

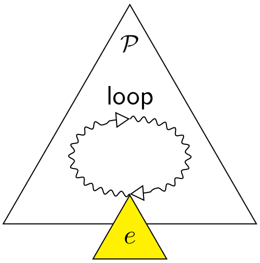
$$\Gamma, \delta \vdash \mathbf{S}(\mathcal{P},\, e,\, m_1) \rightarrowtriangle^* S$$

to some state S, we have either:

1. *$S = \mathbf{S}(\mathcal{P},\, [\nu]_\Omega,\, m_2)$ and the postcondition $(Q\,\nu)\ \Gamma\ \overline{m_2}\ \delta\ (\mathsf{locals}\ \mathcal{P})\ m_2$ holds*
2. *S can reduce further (no undefined behavior has occurred)*

This definition quantifies over too many reductions and does not provide sufficiently strong guarantees to prove soundness of each inference rule. We discuss three problems of this definition:

1. **Escape of the current context.** We quantify over arbitrary reductions starting in a state whose context is $\mathcal{P}$ and require these reductions to end in a state whose context is $\mathcal{P}$ too. We therefore also quantify over reductions in which execution of the expression e itself is finished, and then in turn a part of the surrounding statement in $\mathcal{P}$ is executed. For example:

   We should thus quantify only over reductions $\Gamma, \delta \vdash \mathbf{S}(\mathcal{P},\, e,\, m_1) \rightharpoondown^{*} S$ that do not traverse in upward direction into the context $\mathcal{P}$. These reductions should solely use the corresponding local stack $\mathsf{locals}\ \mathcal{P}$ of $\mathcal{P}$.

2. **Interleaving of subexpressions.** Due to non-deterministic expression evaluation, we have to account for interleaved reductions of the subexpressions e_1 and e_2 of binary operators such as $e_1 \circledcirc e_2$:

   $$\begin{tikzcd}
   e_1 \arrow[r, "+"] & e_1' & e_1' \arrow[r, "+"] & \cdots & {[v_1]_{\Omega_1}} \arrow[dr, bend left] & \\
   \circledcirc & \circledcirc & \circledcirc & & \circledcirc & {[v_2 \circledcirc v_2]_{\Omega_2 \cup \Omega_2}} \\
   e_2 & e_2 \arrow[r, "+"] & e_2' & \cdots \arrow[r, "+"] & {[v_2]_{\Omega_2}} \arrow[ur, bend right] &
   \end{tikzcd}$$

   The proposed definition only provides guarantees about the end result of executing a single expression in isolation, whereas in fact multiple subexpressions may be executed in parallel.

3. **Deallocated memory.** Deallocated objects are kept in the formal memory (see the definition of free in Definition 5.6.5 on page 85) whereas they disappear from the postcondition (see the rule for $\mathsf{alloc}_\tau\ e$ in Definition 8.4.4 and the rule for local variables $\mathsf{local}_\tau\ s$ in Definition 8.5.3). For example, we have the following reduction corresponding to $\{\mathsf{emp}\}\ \mathsf{local}_\tau\ \mathsf{skip}\ \{\mathsf{emp}\}$:

   $$\begin{tikzcd}
   \emptyset \arrow[r, "\text{enter block}"] \arrow[d, dotted, no head] & \{o \mapsto (\mathsf{false},\, \mathsf{new}_\Gamma^{\Diamond(0,1)}\ \tau)\} \arrow[r, "\text{leave block}"] \arrow[d, dotted, no head] & \{o \mapsto \tau\} \arrow[d, dotted, no head] \\
   \{\mathsf{emp}\} & \{\mathsf{top}_o\, \Gamma \xmapsto[\mathsf{false}]{\Diamond(0,1)} - : \tau\} & \{\mathsf{emp}\}
   \end{tikzcd}$$

   The top row denotes the actual memory states whereas the bottom row denotes the corresponding assertion. The last step contains a mismatch.

In order to deal with the issue of context escape, we define a variant of the small-step operational semantics $\Gamma, \delta, \rho \vdash S_1 \rightharpoondown S_2$ in which the local stack is extended with a part ρ described by the precondition. We first define a variant of the function $\mathsf{locals} : \mathsf{ctx} \to \mathsf{stack}$ that yields the corresponding stack (Definition 6.5.6 on page 117).

Definition 8.6.2. *The function* $\mathsf{locals}_\rho : \mathsf{ctx} \to \mathsf{stack}$ *yields the* ρ-extended corresponding local stack *of a program context. It is defined as:*

$$\begin{aligned}
\mathsf{locals}_\rho\ \varepsilon &:= \rho \\
\mathsf{locals}_\rho\ (\mathcal{S}_\mathsf{s}\,\mathcal{P}) &:= \mathsf{locals}_\rho\ \mathcal{P} \\
\mathsf{locals}_\rho\ ((\mathsf{local}_{o:\tau}\,\Box)\,\mathcal{P}) &:= (o,\,\tau)\,(\mathsf{locals}_\rho\ \mathcal{P}) \\
\mathsf{locals}_\rho\ ((\mathcal{S}_\mathsf{e}, e)\,\mathcal{P}) &:= \mathsf{locals}_\rho\ \mathcal{P} \\
\mathsf{locals}_\rho\ ((\mathsf{resume}\ \mathcal{E})\,\mathcal{P}) &:= \varepsilon \\
\mathsf{locals}_\rho\ ((\mathsf{params}\ f\ \overrightarrow{o\tau})\,\mathcal{P}) &:= \overrightarrow{o\tau}\,(\mathsf{locals}_\rho\ \mathcal{P})
\end{aligned}$$

The difference with Definition 6.5.6 is the $\mathsf{locals}_\rho\ \varepsilon$ clause, which yields ρ instead of the empty stack. Note that we do not have $\mathsf{locals}_\rho\ \mathcal{P} = \rho\,(\mathsf{locals}\ \mathcal{P})$.

Definition 8.6.3. *The* ρ-extended small-step reduction $\Gamma, \delta, \rho \vdash S_1 \twoheadrightarrow S_2$ *is a variant of* $\Gamma, \delta \vdash S_1 \twoheadrightarrow S_2$ *(Definition 6.5.8) in which the following rules are modified:*

2a) $\Gamma, \delta, \rho \vdash \mathbf{S}(\mathcal{P},\, \mathcal{E}[\,e_1\,],\, m_1) \twoheadrightarrow \mathbf{S}(\mathcal{P},\, \mathcal{E}[\,e_2\,],\, m_2)$
for any e_2 *and* m_2 *with* $\Gamma, \mathsf{locals}_\rho\ \mathcal{P} \vdash (e_1, m_1) \twoheadrightarrow_\mathsf{h} (e_2, m_2)$

2c) $\Gamma, \delta, \rho \vdash \mathbf{S}(\mathcal{P},\, \mathcal{E}[\,e\,],\, m) \twoheadrightarrow \mathbf{S}(\mathcal{P},\, \overline{\mathsf{undef}}\,(\lightning\,\mathcal{E}\langle e\rangle),\, m)$
if e *is a redex with* $\Gamma, \mathsf{locals}_\rho\ \mathcal{P} \nvdash (m, e)\ \mathsf{safe}$

Fact 8.6.4. *We have* $\Gamma, \delta, \epsilon \vdash S_1 \twoheadrightarrow S_2$ *iff* $\Gamma, \delta \vdash S_1 \twoheadrightarrow S_2$.

Fact 8.6.5. *We have* $\Gamma, \delta, \mathsf{locals}_\rho\ \mathcal{P} \vdash \mathbf{S}(\mathcal{P}_1,\, \phi_1,\, m_1) \twoheadrightarrow \mathbf{S}(\mathcal{P}_2,\, \phi_2,\, m_2)$ *iff:*

$$\Gamma, \delta, \rho \vdash \mathbf{S}(\mathcal{P}\,\mathcal{P}_1,\, \phi_1,\, m_1) \twoheadrightarrow \mathbf{S}(\mathcal{P}\,\mathcal{P}_2,\, \phi_2,\, m_2).$$

In order to deal with the issue of deallocated memory, we define an erasure function that removes deallocated memory objects. We require the pre- and postcondition to hold in the erased version $|m|$ of the memory m.

Definition 8.6.6. *The* deallocation erasure $|m|$ of a memory m *is defined as:*

$$|m| := \lambda o\,.\begin{cases} (w,\,\mu) & \text{if } m\,o = (w,\,\mu) \\ \bot & \text{if } m\,o = \tau \text{ or } m\,o = \bot \end{cases}$$

Since the definitions of $\models_{\Gamma,\delta} \{P\}\,e\,\{Q\}$ and $J, R, T \models_{\Gamma,\delta} \{P\}\,s\,\{Q\}$ have a lot in common, we define a more general judgment to factor out similarities:

$$\Gamma, \delta, \rho \models_n (\mathcal{P},\, \phi,\, m : \Delta)\ \{\hat{Q}\}$$

The intuitive meaning of this judgment is that all $\Gamma, \Delta, \rho \vdash \mathbf{S}(\mathcal{P},\, \phi,\, m \cup m_f) \twoheadrightarrow^* S$ reductions of at most n steps satisfy the following properties:

- They do not get stuck and do never reach an $\overline{\mathsf{undef}}$ state.
- The end-state satisfies the *generalized postcondition* $\hat{Q}$, which ensures that the actual postcondition holds as well as that the end-result has the correct shape and is well-typed (see Definitions 8.6.8 and 8.6.10).

- The *framing memory* m_f may be changed arbitrarily during each individual step to deal with interleaving of subexpressions. The framing memory m_f accounts for memory that has been framed out using the frame rule as well as memory that belongs to other subexpressions.

Definition 8.6.7. *Let* $\hat{Q} : \mathsf{env} \to \mathsf{memenv} \to \mathsf{funenv} \to \mathsf{stack} \to \mathsf{focus} \to \mathsf{mem} \to \mathsf{Prop}$ *be an upward closed predicate under weakening of typing environments and memory typing environments. The judgment* $\Gamma, \delta, \rho \models_n (\mathcal{P},\ \phi,\ m : \Delta)\ \{\hat{Q}\}$ *is inductively defined as:*

$$\frac{}{\Gamma, \delta, \rho \models_0 (\mathcal{P},\ \phi,\ m : \Delta)\ \{\hat{Q}\}} \qquad \frac{\hat{Q}\ \Gamma\ \Delta\ \delta\ \rho\ \phi\ m}{\Gamma, \delta, \rho \models_n (\varepsilon,\ \phi,\ m : \Delta)\ \{\hat{Q}\}}$$

$$\frac{\begin{array}{l} \forall \Delta'\, n'\, m_f \,.\, \Delta \Rrightarrow \Delta' \to n > n' \to m \perp m_f \to \Gamma, \Delta' \vdash m \cup m_f \to \\ \qquad (1)\ \ \exists S' \,.\, \Gamma, \delta, \rho \vdash \mathbf{S}(\mathcal{P},\ \phi,\ m \cup m_f) \rightarrowtriangle S' \\ \qquad (2)\ \ \forall \mathcal{P}'\, \phi'\, m' \,.\, \Gamma, \delta, \rho \vdash \mathbf{S}(\mathcal{P},\ \phi,\ m \cup m_f) \rightarrowtriangle \mathbf{S}(\mathcal{P}',\ \phi',\ m') \to \\ \qquad\qquad\qquad \mathsf{dom}\ m' \setminus \mathsf{dom}\ (m \cup m_f) \cap \mathsf{dom}\ \Delta' = \emptyset \to \exists m''\, \Delta'' \,. \\ \qquad\qquad\qquad (\mathrm{a})\ \ m'' \perp m_f \\ \qquad\qquad\qquad (\mathrm{b})\ \ m' = m'' \cup m_f \\ \qquad\qquad\qquad (\mathrm{c})\ \ \neg\exists \phi_U \,.\, \phi' = \overline{\mathsf{undef}}\, \phi_U \\ \qquad\qquad\qquad (\mathrm{d})\ \ (\forall e \,.\, \phi' = e \to \mathsf{locks}\ e \subseteq \mathsf{locks}\ m'') \\ \qquad\qquad\qquad (\mathrm{e})\ \ \Delta' \Rrightarrow \Delta'' \\ \qquad\qquad\qquad (\mathrm{f})\ \ \Gamma, \Delta'' \vdash m' \\ \qquad\qquad\qquad (\mathrm{g})\ \ \forall \Delta''' \,.\, \Delta' \Rrightarrow \Delta''' \to \Gamma, \Delta''' \vdash m' \to \Delta'' \Rrightarrow \Delta''' \\ \qquad\qquad\qquad (\mathrm{h})\ \ \Gamma, \delta, \rho \models_{n'} (\mathcal{P}',\ \phi',\ m'' : \Delta'')\ \{\hat{Q}\} \end{array}}{\Gamma, \delta, \rho \models_n (\mathcal{P},\ \phi,\ m : \Delta)\ \{\hat{Q}\}}$$

Definition 8.6.8. *The* generalized postcondition $\hat{Q}_{\tau_{\mathsf{lr}}} : \mathsf{env} \to \mathsf{memenv} \to \mathsf{funenv} \to \mathsf{stack} \to \mathsf{focus} \to \mathsf{mem} \to \mathsf{Prop}$ of an expression with postcondition $Q : \mathsf{lrval} \to \mathsf{assert}$ and type τ_{lr} *is defined as:*

$$\begin{array}{rl} \hat{Q}_{\tau_{\mathsf{lr}}} := \lambda \Gamma\ \Delta\ \delta\ \rho\ \phi\ m \,.\, \exists \nu\, \Omega \,. & (1)\ \ \phi = [\nu]_\Omega \\ & (2)\ \ \Gamma, \Delta \vdash \nu : \tau_{\mathsf{lr}} \\ & (3)\ \ \mathsf{locks}\ m = \Omega \\ & (4)\ \ (Q\, \nu)\ \Gamma\ \Delta\ \delta\ \rho\ |m|. \end{array}$$

Definition 8.6.9. *The judgment* $\models_{\Gamma,\delta} \{P\}\, e\, \{Q\}$ *of* partial correctness of an expression e *is defined as follows. For each* $n \in \mathbb{N}$, $\Gamma \subseteq \Gamma'$, Δ, $\delta \subseteq \delta'$, m, ρ *and* τ_{lr} *with* $\vdash \Gamma'$ *and* $\Gamma', \Delta \vdash \delta'$ *and* $\Gamma', \Delta \vdash m$ *and* $\mathsf{locks}\ m = \emptyset$ *and* $\Delta \vdash \rho$ *and* $\Gamma', \Delta, \mathsf{types}\ \rho \vdash e : \tau_{\mathsf{lr}}$ *and* $\mathsf{locks}\ e = \emptyset$*:*

$$P\ \Gamma'\ \Delta\ \delta'\ \rho\ |m| \quad \text{implies} \quad \Gamma', \delta', \rho \models_n (\varepsilon,\ e,\ m : \Delta)\ \{\hat{Q}_{\tau_{\mathsf{lr}}}\}.$$

The judgment $\Gamma, \delta, \rho \models_n (\mathcal{P},\ \phi,\ m : \Delta)\ \{\hat{Q}\}$ uses step-indexing [AM01] so we can conveniently prove properties by induction on the number of steps n. Step-indexing is essential to deal with (mutually) recursive functions in Section 8.7.3.

The judgment $\Gamma, \delta, \rho \models_n (\mathcal{P},\ \phi,\ m : \Delta)\ \{\hat{Q}\}$ quantifies over reductions $\Gamma, \delta, \rho \vdash \mathbf{S}(\mathcal{P},\ \phi,\ m \cup m_f) \twoheadrightarrow \mathbf{S}(\mathcal{P}',\ \phi',\ m')$ in which the initial memory m is extended with an arbitrary framing part m_f for which we have $\Gamma, \Delta' \vdash m \cup m_f$. The side-condition $\mathsf{dom}\ m' \setminus \mathsf{dom}\ (m \cup m_f) \cap \mathsf{dom}\ \Delta' = \emptyset$ ensures that no object identifiers are used that are already in use by Δ'.

After each reduction step $\Gamma, \delta, \rho \vdash \mathbf{S}(\mathcal{P},\ \phi,\ m \cup m_f) \twoheadrightarrow \mathbf{S}(\mathcal{P}',\ \phi',\ m')$ we have to establish that the resulting memory has the shape $m' = m'' \cup m_f$. Moreover, we have to show that there exists a memory environment Δ'' with $\Gamma, \Delta'' \vdash m'$. The condition $\Delta' \Rightarrow \Delta''$ ensures that Δ'' is forward (see Definition 6.6.8 on page 122) and the condition $\forall \Delta''' .\, \Delta' \Rightarrow \Delta''' \to \Gamma, \Delta''' \vdash m' \to \Delta'' \Rightarrow \Delta'''$ ensures that Δ'' is the least forward memory environment. The condition $\mathsf{locks}\ e \subseteq \mathsf{locks}\ m''$ ensures that the accumulated locks are separated.

The judgment $\bar{P} \models_{\Gamma,\delta} s$ for statements is defined in the same way as the expression judgment. Recall that the function $\bar{P} : \mathsf{direction} \to \mathsf{assert}$ bundles the conditions of the judgment $J, R, T \vdash_{\Gamma,\delta} \{P\}\, s\, \{Q\}$ (see Definition 8.5.4).

Definition 8.6.10. *The* generalized postcondition $\hat{Q}_{s:(\beta,\ \tau^?)} : \mathsf{env} \to \mathsf{memenv} \to \mathsf{stack} \to \mathsf{focus} \to \mathsf{mem} \to \mathsf{Prop}$ of a statement with postcondition Q and type $(\beta,\ \tau^?)$ *is defined as:*

$$\begin{aligned}\hat{Q}_{s:(\beta,\ \tau^?)} := \lambda \Gamma\ \Delta\ \delta\ \rho\ \phi\ m \,.\, \exists d \,.\, &(1)\ \phi = (d,\ s)\\ &(2)\ (d,\ s)\ \mathsf{out}\\ &(3)\ \Gamma, \Delta \vdash d : (\beta,\ \tau^?)\\ &(4)\ \mathsf{locks}\ m = \emptyset\\ &(5)\ (\bar{P}\, d)\ \Gamma\ \Delta\ \delta\ \rho\ |m|.\end{aligned}$$

The predicate $(d,\ s)$ out *is defined in Definition 6.5.7 on page 118. It states that traversal through the zipper goes in outward direction.*

Definition 8.6.11. *The judgment* $\bar{P} \models_{\Gamma,\delta} s$ *of* partial correctness of a statement s *is defined as follows. For each* $n \in \mathbb{N}$, $\Gamma \subseteq \Gamma'$, Δ, $\delta \subseteq \delta'$, m, ρ, d *and* $(\beta,\ \tau^?)$ *with* $\vdash \Gamma'$ *and* $\Gamma', \Delta \vdash \delta'$ *and* $\Gamma', \Delta \vdash m$ *and* $\mathsf{locks}\ m = \emptyset$ *and* $\Delta \vdash \rho$ *and* $\Gamma', \Delta, \mathsf{types}\ \rho \vdash s : (\beta,\ \tau^?)$ *and* $(d,\ s)$ in*:*

$$(\bar{P}\, d)\ \Gamma'\ \Delta\ \delta'\ \rho\ |m| \quad \text{implies} \quad \Gamma', \delta', \rho \models_n (\varepsilon,\ (d,\ s),\ m : \Delta)\ \{\hat{Q}_{s:(\beta,\ \tau^?)}\}.$$

The predicate $(d,\ s)$ in *is defined in Definition 6.5.7 on page 118. It states that traversal through the zipper goes in upward direction.*

The above definition shows the advantage of bundling the conditions as a function $\bar{P} : \mathsf{direction} \to \mathsf{assert}$. Instead of having to separately treat 8 combinations corresponding to the directions $\{\searrow, \curvearrowright\} \times \{\nearrow, \upuparrows, \uparrow, \curvearrowright\}$ in which a statement can be entered and exited, we have just one case that covers all of these combinations.

Our approach extends easily to other forms of non-local control flow. For example, in the Coq development we have added an (unstructured) **`switch`** statement, and that had little impact on the soundness proof of the separation logic.

We now show that the judgment $\bar{P} \models_{\Gamma,\delta} s$ enjoys the intended property of partial program correctness.

Theorem 8.6.12 (Adequacy)**.** *Suppose we have* $\vdash \Gamma$ *and* $\Gamma, \overline{m} \vdash \delta$ *and* $\Gamma \vdash m$ *and* $\mathsf{locks}\ m = \emptyset$ *and* $\Gamma, \overline{m}, \varepsilon \vdash s : (\tau^?, \beta)$*. Now if:*

$$R, \mathsf{False}, \mathsf{False} \models_{\Gamma,\delta} \{P\}\, s\, \{Q\} \quad \text{and} \quad P\, \Gamma\, \overline{m}\, \delta\, \varepsilon\, |m|$$

then for each reduction $\Gamma, \delta \vdash \mathbf{S}(\varepsilon, (\searrow, s), m) \twoheadrightarrow^* S$ *we have either:*

1. *The postcondition holds:* $S = \mathbf{S}(\varepsilon, (\searrow, s), m')$ *with* $Q\, \Gamma\, \overline{m'}\, \delta\, \varepsilon\, |m'|$*.*
2. *The returning condition holds:* $S = \mathbf{S}(\varepsilon, (\Uparrow v, s), m')$ *with* $R\, v\, \Gamma\, \overline{m'}\, \delta\, \varepsilon\, |m'|$*.*
3. *Further reduction is possible:* $\Gamma, \delta \vdash S \twoheadrightarrow S'$ *for some* S'*.*

Theorem 8.6.13 (Soundness of the separation logic)**.** *We have:*

$$\begin{array}{lcl} \vdash_{\Gamma,\delta} \{P\}\, e\, \{Q\} & \text{implies} & \models_{\Gamma,\delta} \{P\}\, e\, \{Q\} \\ P \vdash_{\Gamma,\delta} s & \text{implies} & P \models_{\Gamma,\delta} s \end{array}$$

The above theorem is proven by induction on the derivations of $\vdash_{\Gamma,\delta} \{P\}\, e\, \{Q\}$ (Definition 8.4.4) and $J, R, P \vdash_{\Gamma,\delta} \{T\}\, s\, \{Q\}$ (Definition 8.5.3), so we have to show that each inference rule holds in the interpretation. For the details about these proofs, we refer to the Coq development, but we discuss some key points here:

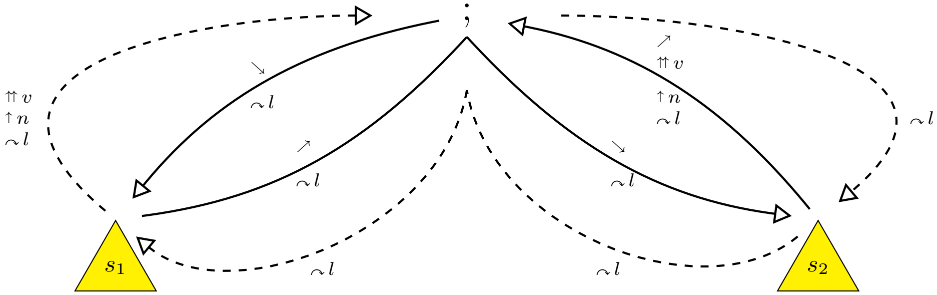
- We have proven abstract versions of the frame and weakening rule for the judgment $\Gamma, \delta, \rho \models_n (\mathcal{P}, \phi, m : \Delta)\ \{\hat{Q}\}$. The frame and weakening rule for the expression and statement judgment are corollaries of these more general rules.
- In order to prove the inference rules of binary expressions constructs we have to account for interleaving of the subexpression. We have proven a more general result for the judgment $\Gamma, \delta, \rho \models_n (\mathcal{P}, \phi, m : \Delta)\ \{\hat{Q}\}$.
- The inference rules for statements involve quite some cases due to non-local control flow. Using the (d, s) in and (d, s) out predicates we avoid a blowup in the number of cases we have to consider.

We consider the rule for composition as an example.

Lemma 8.6.14. *The rule for composition holds:*

$$\frac{R, J, T \models_{\Gamma,\delta} \{P\}\, s_1\, \{P'\} \qquad R, J, T \models_{\Gamma,\delta} \{P'\}\, s_2\, \{Q\}}{R, J, T \models_{\Gamma,\delta} \{P\}\, s_1\, ;\ s_2\, \{Q\}}$$

Proof. This lemma is proven by induction on the number of steps and involves chasing all possible reduction paths. The following reduction paths may occur:



□

It is important to note that one does not need to trust the construction of the judgments $\vdash_{\Gamma,\delta} \{P\}\, e \,\{Q\}$ and $J, R, P \vdash_{\Gamma,\delta} \{T\}\, s \,\{Q\}$ in order to believe that our separation logic is indeed sound. One only has to trust the statements of Theorems 8.6.12 and 8.6.13.

8.7 Extensions of the separation logic

In the previous sections we have omitted some features of our separation logic to make the presentation more comprehensible. This section describes the features that are left out, but that have been formalized in Coq. These features make our separation logic more powerful and more practical to use. The separation logic and proofs described in the previous sections are an instance of the full system formalized in Coq.

8.7.1 Expression invariants

We extend the expression judgment $\vdash_{\Gamma,\delta} \{P\}\, e \,\{Q\}$ with a *shared invariant* B that can be used by all subexpressions:

$$B \vdash_{\Gamma,\delta} \{P\}\, e \,\{Q\}$$

The key feature of this extension is that one can shift all read-only memory into the shared invariant B, and use the pre- and postcondition solely for writable memory. One thus does not have to subdivide the entire pre- and post condition into two parts at each occurrence of a binary expression construct. The pre- and post condition can thus be emp for expressions without assignments and function calls.

This extension involves modifications of the expression judgment (Definition 8.4.4) as well as an adaptation of the soundness proof. We present some representative rules in this thesis and refers to the Coq development for all details and proofs.

The following frame rule can be used to shift a part of the shared invariant into the pre- and postcondition:

$$\frac{A * B \vdash_{\Gamma,\delta} \{P\}\, e \,\{Q\}}{B \vdash_{\Gamma,\delta} \{A * P\}\, e \,\{A * Q\}}$$

In order to prove an individual subexpression, one may use the shared invariant to show that the result is well-defined. For example:

$$\frac{\begin{array}{c} B \vdash_{\Gamma,\delta} \{P_1\}\, e_1 \,\{Q_1\} \qquad B \vdash_{\Gamma,\delta} \{P_2\}\, e_2 \,\{Q_2\} \\ \forall v_1\, v_2 \,.\, (Q_1\, v_1 * Q_2\, v_2 \models_{\Gamma,\delta} \exists v' \,.\, (B \mathrel{-\!\!*} (v_1 \circledcirc v_2) \Downarrow v') \wedge Q'\, v') \end{array}}{B \vdash_{\Gamma,\delta} \{P_1 * P_2\}\, e_1 \circledcirc e_2 \,\{Q'\}}$$

$$\frac{B \vdash_{\Gamma,\delta} \{P\}\, e \,\{Q\} \qquad \forall p \,.\, (Q\, p \models_{\Gamma,\delta} \exists v' \,.\, (B \mathrel{-\!\!*} (\mathsf{load}\ p) \Downarrow v') \wedge Q'\, v')}{B \vdash_{\Gamma,\delta} \{P\}\, \mathsf{load}\ e \,\{Q'\}}$$

Statement judgments are not extended with shared invariants. The shared invariant should thus be emp in order to prove a statement judgment:

$$\frac{\mathsf{emp} \vdash_{\Gamma,\delta} \{P\}\, e \,\{\lambda\_ \,.\, Q \,\Diamond\}}{R, J, T \vdash_{\Gamma,\delta} \{P\}\, e \,\{Q\}}$$

In order to prove soundness, we have defined a variant $B \models_{\Gamma,\delta} \{P\}\, e\, \{Q\}$ of the judgment for partial program correctness of expressions (Definition 8.6.9). The judgment $B \models_{\Gamma,\delta} \{P\}\, e\, \{Q\}$ quantifiers over reductions in which the memory is subdivided in three parts $m \cup m_B \cup m_f$. These parts are used as follows:

- The part m belongs exclusively to the subexpression e and is thus fully controlled by the precondition P and postcondition Q.
- The part m_B is controlled by the shared invariant B. This part may change arbitrarily during each reduction step of e and during each interleaved reduction step of any other subexpression. The sole requirement is that the shared invariant B remains to hold.
- The framing part m_f corresponds to memory controlled by other subexpressions and memory that has been framed out. Other subexpressions may change this part arbitrarily, but e should preserve it as if.

8.7.2 Simple expressions

The postcondition Q of the expression judgment $B \vdash_{\Gamma,\delta} \{P\}\, e\, \{Q\}$ of our separation logic is a function $Q : \mathsf{lrval} \to \mathsf{assert}$ that ranges over return values. The postcondition Q can thus capture the results of expressions whose value cannot be determined from just the precondition P. Although this approach is very general, it comes at the cost of making the inference rules (Definition 8.4.4) rather complicated with an abundance of universal and existential quantifiers.

For all expression constructs apart from the $\mathsf{alloc}_\tau\ e$ construct and function calls (which we treat in Section 8.7.3), the resulting value *can* actually be determined from just P. We therefore define a variant of the expression judgment.

Definition 8.7.1. *The judgment $B \vdash_{\Gamma,\delta} \{P\}\, e\, \{\nu \mid Q\}$ is defined as follows:*

$$B \vdash_{\Gamma,\delta} \{P\}\, e\, \{\nu \mid Q\} := B \vdash_{\Gamma,\delta} \{P\}\, e\, \{\lambda \nu' \,.\, \ulcorner \nu' = \nu \urcorner * Q\}$$

Here we have $\nu \in \mathsf{lrval}$ and $Q \in \mathsf{assert}$.

The postcondition Q now no longer binds the resulting value. Albeit less general, this variant of the expression judgment has much simpler derivation rules.

Lemma 8.7.2. *We have the following admissible rules:*

$$\frac{B * Q_1 * Q_2 \models_{\Gamma,\delta} v_1 \circledcirc v_2 \Downarrow v' \qquad B \vdash_{\Gamma,\delta} \{P_1\}\, e_1\, \{v_1 \mid Q_1\} \qquad B \vdash_{\Gamma,\delta} \{P_2\}\, e_2\, \{v_2 \mid Q_2\}}{B \vdash_{\Gamma,\delta} \{P_1 * P_2\}\, e_1 \circledcirc e_2\, \{v' \mid Q_1 * Q_2\}}$$

$$\frac{B * Q \models_{\Gamma,\delta} \mathsf{load}\ a \Downarrow v' \qquad B \vdash_{\Gamma,\delta} \{P\}\, e\, \{a \mid Q\}}{B \vdash_{\Gamma,\delta} \{P\}\, \mathsf{load}\ e\, \{v' \mid Q\}}$$

Proof. These rules follow from the rules of the judgment $B \vdash_{\Gamma,\delta} \{P\}\, e\, \{Q\}$. □

In the Coq development we have proven similar rules for all expression constructs apart from function calls and the $\mathsf{alloc}_\tau\ e$ construct.

8.7.3 Function calls

In previous work [Kre14a] we extended the judgments of our separation logic with an environment of function specifications that assigns pre- and postconditions to function names. This approach is adapted from Von Oheimb [Ohe01, vON02] and involves the following two inference rules:

- The first rule allows one to extend the environment with a collection of mutually recursive functions. One has to prove that each function is correct with respect to its respective pre- and postcondition.
- The second rule allows one to call a function that is in the environment. This function has thus already been proven correct using the first rule.

In this thesis we pursue a more modern approach based on higher-order separation logic. Higher-order separation logic [BBT05, BJSB11] allows function specifications as part of the assertions. This extension requires a modified definition of assertions (Definition 8.2.1) and extends the expression judgment (Definition 8.4.4) with a rule for function calls.

As usual in higher-order separation logic, we use step-indexed assertions [AM01, AMRV07] to deal with mutually recursive functions. Assertions $P, Q \in \mathsf{assert}$ become predicates over natural numbers, where the natural number counts the number of remaining reduction steps:

$$P, Q \in \mathsf{env} \to \mathsf{memenv} \to \mathsf{funenv} \to \mathsf{stack} \to \mathbb{N} \to \mathsf{mem} \to \mathsf{Prop}$$

We extend the assertion language with the following assertion that describes the specification of a function f of type $\vec{\tau} \to \tau$:

$$(f^{\vec{\tau} \mapsto \tau} \mapsto \forall \vec{y}\, \vec{v} \,.\, \{P\, \vec{y}\, \vec{v}\}\ \{Q\, \vec{y}\, \vec{v}\})$$

Universal quantification over the function parameters $\vec{v}$ and logical variables $\vec{y}$ can be used to relate the precondition $P\, \vec{y}\, \vec{v} : \mathsf{assert}$ and postcondition $Q\, \vec{y}\, \vec{v} : \mathsf{val} \to \mathsf{assert}$. The pre- and postcondition should furthermore be stack independent (that is, $P\, \vec{y}\, \vec{v} \models_{\Gamma,\delta} (P\, \vec{y}\, \vec{v}) \uparrow$ and likewise for Q) because local variables will have a different meaning at the caller than at callee.

In order to prove the specification of a function f, one has to establish that the function body $\delta\, f$ is correct with respect to the precondition $P\, \vec{y}\, \vec{v}$ and postcondition $Q\, \vec{y}\, \vec{v}$ for all values of the function parameters $\vec{v}$ and for all instantiations of the logical variables $\vec{y}$. The inference rule is as follows:

$$\frac{\Gamma\, f = (\vec{\tau}, \tau) \qquad \delta\, f = s \qquad \forall \vec{y}\, \vec{v} \,.\, \big(\vdash_{\Gamma,\delta} \{\circledast_i [\, x_i \xmapsto[\mathsf{false}]{\Diamond(0,1)} |\, v_i\,|_\circ : \tau_i\,] * P\}\, s\, \{\circledast_i [\, x_i \xmapsto[\mathsf{false}]{\Diamond(0,1)} - : \tau_i\,] * Q\}\big)}{\mathsf{emp} \models_{\Gamma,\delta} (f^{\vec{\tau} \mapsto \tau} \mapsto \forall \vec{y}\, \vec{v} \,.\, \{P\, \vec{y}\, \vec{v}\}\ \{Q\, \vec{y}\, \vec{v}\})}$$

where we use the following shorthand for $P \in \mathsf{assert}$ and $Q : \mathsf{val} \to \mathsf{assert}$:

$$\vdash_{\Gamma,\delta} \{P\}\, s\, \{Q\} \quad := \quad Q, (\lambda l \,.\, \mathsf{False}), (\lambda n \,.\, \mathsf{False}) \vdash_{\Gamma,\delta} \{P\}\, s\, \{Q\, \mathsf{nothing}\}$$

We use the *later modality* $\triangleright$ to reason about (mutually) recursive functions [Nak00, AMRV07, BJSB11]. This modality enjoys the following laws, where the middle is the so called Löb rule:

$$\frac{}{P \models_{\Gamma,\delta} \triangleright P} \qquad \frac{}{\triangleright P \to P \models_{\Gamma,\delta} P} \qquad \frac{}{\triangleright (P \wedge Q) \models_{\Gamma,\delta} \triangleright P \wedge \triangleright Q}$$

The assertion $\triangleright P$ describes that P holds after a step of execution. When calling a function, it is sufficient that the function specification holds later:

$$\frac{\begin{array}{c} B \vdash_{\Gamma,\delta} \{P\}\, e\, \{Q\, \Diamond\} \qquad \forall i\,.\, \big(B \vdash_{\Gamma,\delta} \{P_i\}\, e_i\, \{Q_i\, \Diamond\}\big) \\ \forall v\, \vec{v}\,.\, \Big(B * Q\, v * \circledast_i[\, Q_i\, v_i\,] \models_{\Gamma,\delta} \exists f\,.\, \ulcorner v = \mathsf{ptr}\, f^{\vec{\tau} \mapsto \tau} \urcorner * \\ \qquad\qquad \triangleright (f^{\vec{\tau}\mapsto\tau} \mapsto \forall \vec{y}\,\vec{v}\,.\, \{P_f\, \vec{y}\, \vec{v}\}\ \{Q_f\, \vec{y}\, \vec{v}\}) * \\ \qquad\qquad P_f\, \vec{y}\, \vec{v} * \big(\forall v_r\,.\, Q_f\, \vec{y}\, \vec{v}\, v_r \mathrel{-\!\!*} (B * Q'\, v_r)\big)\Big) \end{array}}{B \vdash_{\Gamma,\delta} \{P * \circledast_i[\, P_i\,]\}\, e(\vec{e})\, \{Q'\}}$$

The rule for calling a function allows one to use the shared invariant B to prove the precondition $P_f\, \vec{y}\, \vec{v}$, and in the end, the shared invariant B has to be reobtained from the postcondition $Q_f\, \vec{y}\, \vec{v}\, v_r$. The invariant allows one to describe shared-memory among multiple function calls in the same expression (for example a buffer or hashtable). Note that our axiomatic semantics is not complete. Consider:

```
int x = 0;
int f(int y) { return (x = y); }
int main() { f(3) + f(4); return x; }
```

Since the invariant B should hold before, after, and in between each function call in the expression `f(3) + f(4)`, the best choice for it is $\mathtt{x} \mapsto 0 \vee \mathtt{x} \mapsto 3 \vee \mathtt{x} \mapsto 4$. Hence, one can only prove that the program returns 0, 3 or 4, whereas it is actually guaranteed to return always 3 or 4.

8.8 Example: verification of gcd

In order to demonstrate our separation logic, we verify Euclid's algorithm for computing the greatest common divisor. The C code of this algorithm is as follows:

```
int gcd(unsigned int y, unsigned int z) {
  l: if (z) {
    int tmp;
    z = (tmp = y % z, y = z, tmp);
    goto l;
  }
  return y;
}
```

The code is written in a non-idiomatic way to demonstrate interesting features of our semantics. It contains a `goto` that jumps out of a scope as well as an expression with multiple side-effects. We verify the function body without the `return` statement.

Notation 8.8.1. *We abbreviate* $\mathsf{uint} := \mathsf{unsigned\ int}$.

Theorem 8.8.2. *The following Hoare triple is derivable:*

```
$R, J, T \vdash_{\Gamma,\delta} \{(x_0 \mapsto y : \mathsf{uint}) * (x_1 \mapsto z : \mathsf{uint})\}$
            $l : \mathsf{if}\ (\mathsf{load}\ x_1)\ ($
              $\mathsf{local}_{\mathsf{uint}}\ ($
                $x_2 := (x_0 := (\mathsf{load}\ x_1\ \%\ \mathsf{load}\ x_2),\ x_1 := \mathsf{load}\ x_2,\ \mathsf{load}\ x_0)\,;$
                $\mathsf{goto}\ l$
              $)$
            $)\ \mathsf{else\ skip}$
          $\{(x_0 \mapsto (\mathsf{gcd}\ y\ z) : \mathsf{uint}) * (x_1 \mapsto 0 : \mathsf{uint})\}$
```

provided $J\,l = \exists y'\,z'\,.\,(\mathsf{gcd}\ y'\ z' = \mathsf{gcd}\ y\ z * (x_0 \mapsto y' : \mathsf{uint}) * (x_1 \mapsto z' : \mathsf{uint}))$.

Proof. An outline of the derivation of the Hoare triple is as follows:

```
$\{(x_0 \mapsto y : \mathsf{uint}) * (x_1 \mapsto z : \mathsf{uint})\}$
  $l :$
$\{J\,l\}$
$\{\mathsf{gcd}\ y'\ z' = \mathsf{gcd}\ y\ z * (x_0 \mapsto y' : \mathsf{uint}) * (x_1 \mapsto z' : \mathsf{uint})\}$
  $\mathsf{if}\ (\mathsf{load}\ x_1)\ ($
    $\{z' \neq 0 * (x_0 \mapsto y' : \mathsf{uint}) * (x_1 \mapsto z' : \mathsf{uint})\}$
      $\mathsf{local}_{\mathsf{uint}}\ ($
        $\{(x_0 \mapsto - : \mathsf{uint}) * (x_1 \mapsto y' : \mathsf{uint}) * (x_2 \mapsto z' : \mathsf{uint})\}$
          $x_2 := (x_0 := (\mathsf{load}\ x_1\ \%\ \mathsf{load}\ x_2),\ x_1 := \mathsf{load}\ x_2,\ \mathsf{load}\ x_0)\,;$
        $\{(x_0 \mapsto (y'\ \mathsf{mod}\ z') : \mathsf{uint}) * (x_1 \mapsto z' : \mathsf{uint}) * (x_2 \mapsto (y'\ \mathsf{mod}\ z') : \mathsf{uint})\}$
        $\{J\,l\}$
          $\mathsf{goto}\ l$
        $\{(x_0 \mapsto - : \mathsf{uint}) * (x_1 \mapsto (\mathsf{gcd}\ y\ z) : \mathsf{uint}) * (x_2 \mapsto 0 : \mathsf{uint})\}$
      $)$
    $\{(x_0 \mapsto (\mathsf{gcd}\ y\ z) : \mathsf{uint}) * (x_1 \mapsto 0 : \mathsf{uint})\}$
  $)\ \mathsf{else}\ ($
    $\{z' = 0 * (x_0 \mapsto y' : \mathsf{uint}) * (x_1 \mapsto z' : \mathsf{uint})\}$
      $\mathsf{skip}$
    $\{\mathsf{gcd}\ y'\ 0 = \mathsf{gcd}\ y\ z * (x_0 \mapsto y' : \mathsf{uint}) * (x_1 \mapsto 0 : \mathsf{uint})\}$
  $)$
$\{(x_0 \mapsto (\mathsf{gcd}\ y\ z) : \mathsf{uint}) * (x_1 \mapsto 0 : \mathsf{uint})\}$
```

□

In the proof, we only displayed logical conditions (marked blue) in the precondition where they occurred first and have treated them implicitly consecutively. In the Coq development we have extended the separation logic with the following rule to make this informal treatment of logical conditions formal.

$$\frac{\forall l \in \mathsf{labels}\ s\,.\,(J\,l \models_{\Gamma,\delta} \ulcorner A \urcorner * \mathsf{True}) \qquad A \to (R, J, T \vdash_{\Gamma,\delta} \{P\}\, s\, \{Q\})}{R, J, T \vdash_{\Gamma,\delta} \{\ulcorner A \urcorner * P\}\, s\, \{Q\}}$$

The Coq proof of Theorem 8.8.2 amounts to 115 lines of code. We currently apply the inference rules of the separation logic by hand and hardly use any form of proof automation. See also the discussion of future work in Section 11.2.4.

Chapter 9

Formalization in Coq

Real-world programming language have a large number of features that require large formal descriptions. As this thesis has shown, the C programming language is not different in this regard. On top of that, the C semantics is very subtle due to an abundance of delicate corner cases. Designing a semantics for C and proving properties about such a semantics therefore inevitably requires computer support.

For these reasons, we have used the Coq proof assistant [Coq15] to formalize all definitions and theorems in this thesis. Although Coq does not guarantee the absence of mistakes in our definitions, it provides a rigorous set of checks on our definitions. Already Coq's type checking of definitions provides an effective sanity check. On top of that, we have used Coq to prove all metatheoretical results stated in this thesis. These results include correspondences between different kinds of semantics. Last but not least, using Coq's program extraction facility we have extracted an exploration tool to test our semantics on small example programs.

This chapter describes the formalization as well as our Coq support library developed as part of it. This library has a great number of definitions and theory on common data structures such as lists, finite sets, finite maps and hashsets that were essential for the described formalization efforts.

9.1 The Coq proof assistant

The Coq proof assistant [Coq15, BC04] is based on the calculus of inductive constructions [CH88, CP90], a dependent type theory with (co)inductive types. Coq is both a pure functional programming language with an expressive type system, and a language for mathematical statements and proofs. We highlight some aspects of Coq that have been essential to the CH_2O development.

Type classes Type classes [WB89], as popularized by the Haskell functional programming language, provide a means for organization of abstract structures. We have used type classes in Coq [SO08] for a number of purposes:

- Type classes are used to overload commonly used notations such as those for typing judgments (we have 25 different typing judgments).

- Type classes are used to organize algebraic structures such as separation algebras. Also, they are used to organize multiple implementation of abstract data structures such as finite sets and finite maps.
- Type classes are used to parameterize the whole Coq development by an implementation environment (Definition 3.4.1 on page 48).

Notations Coq has an extensible mechanism for defining complex notations. We have used this mechanism in conjunction with unicode symbols and type classes to obtain notations that are close to those used in this thesis.

Program extraction. Coq provides a facility called program extraction that automatically translates a functional program written in Coq into corresponding OCaml code [Pau89, Let04]. We have used extraction to obtain an executable version of the interpreter described in Chapter 7, which itself is written directly in Coq.

We have not used program extraction in the traditional sense where it is used to extract algorithms from proofs. Instead, we have used it on programs that are close to those one would write in an ML-like language. Program extraction just removes logical parts that are computationally irrelevant such as invariants of data structures and termination arguments of functions that are not structurally recursive.

The Ltac tactic language. Coq proofs are built using a number of *tactics*. Coq provides tactics for elementary proof steps (such as ∀-introduction) as well as more advanced procedures such as Prolog-style inference (the `eauto` tactic) and solving Presburger arithmetic (the `lia` tactic). Coq's domain specific language Ltac [Del00] allows one to program more complex tactics and decision procedures. Ltac is used heavily in the CH_2O development, in particular, it is used to automatically discharge 'boring' cases of large proofs by induction.

9.2 The CH_2O support library

A fair amount of operations and theory on data structures common in programming was needed during the development of the CH_2O formalization. Most of these operations and corresponding theory were absent in the Coq standard library.

In particular, the definitions corresponding to the memory model involve an abundance of list surgery to translate between bit sequences and tree representations (see for example Definitions 5.4.16, 5.4.8, 5.5.7 and 5.5.11). Manipulation of finite maps is frequent to define the memory operations (see Definition 5.6.5) and the separation algebra operations on memories (see Definition 8.1.6). The formalization also involves monadic programming: the set monad is used to compute all behaviors of a program (see Section 6.8), and the combined error state monad is used in the translation from CH_2O abstract C into CH_2O core C (see Section 7.2).

For these reasons, we have developed a support library of with results on common data structures and abstractions used in programming. This library contains ∼12.500 lines of code. Its main parts are as follows:

- **Lists.** This part of the library describes numerous operations on lists. Common properties about these operations, as well as properties about the interaction of these operations, are proven.

  Notable operations: append, length, lookup, update at position, delete at position, alter at position, replicate, resize with padding, take, drop, filter, left fold, right fold, point-wise lifting of functions, lifting of predicates and relations to lists, prefix and suffix inclusion, permutations, treatment of lists as sets, operations on sublists, and the monadic structure of lists.

- **Finite maps.** This part of the library describes an abstract interface for finite maps and provides various implementations of this interface. The abstract interface provides common theory and automation.

  Instances: finite maps represented as lists with Coq's type of unary natural numbers `nat` as keys, finite maps represented as uncompressed radix-2 search trees with Coq's type of binary positive natural numbers `positive`, binary natural numbers `N`, binary integers `Z` and strings `string` as keys, finite maps represented as association lists with an arbitrary lexicographic order as keys.

  Notable operations: empty map, lookup, singleton, insert, delete, a generalized alter operation, point-wise lifting of functions, lifting of predicates and relations to maps, inclusion, operations to combine maps such as disjoint union, intersection and difference, conversion to and from lists, and extraction of the domain of a finite map.

- **Finite sets.** This part of the library describes an abstract interface for finite sets and provides various implementations of this interface.

  Instances: any finite map (with elements of type `unit`), lists, and hashsets based on radix-2 search trees, see also Section 9.2.1.

  Notable operations: empty set, inclusion, union, intersection, difference, size, filter, the monad structure of sets, and choice from non-empty finite sets.

- **Monads.** This part of the library provides overloaded notations for monads, in particular the *do* notation. We do not treat the monad laws, but provide more powerful tactics specific to the option, finite set, and error state monad.

  Instances: option monad, list monad, finite set monad, and a combined error state monad (Definition 7.2.1 on page 136).

We have used type classes to overload notations and to provide abstract interfaces for finite maps and finite sets. Our approach to abstract interfaces is inspired by the *unbundled* approach of Spitters and van der Weegen [SvdW11].

9.2.1 Radix-2 search trees and hashsets

The entire CH$_2$O development makes frequent use of finite sets and finite maps. Since we have to execute parts of the development, we have implemented these using radix-2 search trees to obtain logarithmic time operations. Our implementation of radix-2 search trees is based on CompCert's [Ler09a].

Contrary to CompCert's implementation of finite maps as radix-2 search trees, we ensure that extensional equality and Leibniz equality coincide. This means that given finite maps m_1 and m_2, we have $m_1 = m_2$ iff $m_1\,x = m_2\,x$ for all x. This property does not hold in CompCert's implementation because different trees representing the same finite map are allowed. We have equipped our radix-2 search trees with a proof of canonicity to remedy this shortcoming.

Having an extensional equality on finite sets and finite maps reduces the need for *setoids* (types equipped with an equivalence relation) and therefore reduces formalization overhead. In particular, setoid equality on finite maps and finite sets causes considerable overhead because finite sets appear deeply throughout the CH_2O syntax (for example the locksets Ω in expressions $[\nu]_\Omega$, see Definition 6.1.4 on page 95).

Radix-2 search trees are also used to implement hashsets, which are used to filter out duplicates in the implementation of the reachable state set of the interpreter (see Section 7.4). Since the correctness proofs of operations on hashsets do not rely on any properties of the hash function (in the worst case all elements end up in the same hash bucket, but that just affects efficiency), we can use the efficient OCaml standard library function `Hashtbl.hash` after extraction.

9.2.2 Monads

We have used type classes to provide overloaded notations for monadic operations. For example, we provide Haskell-like notations for the monadic bind:

```
Class MBind (M : Type → Type) :=
  mbind : ∀ {A B}, (A → M B) → M A → M B.
Notation "m ≫= f" := (mbind f m).
Notation "x ← y ; z" := (y ≫= (λ x, z)).
Notation "' ( x1 , x2 ) ← y ; z" :=
  (y ≫= (λ x, let '(x1, x2) := x in z)).
```

Instances are defined in the expected way:

```
Instance option_bind: MBind option := λ A B f x,
  match x with Some a => f a | None => None end.
Instance list_bind : MBind list := λ A B f,
  fix go l := match l with [] => [] | x :: l => f x ++ go l end.
```

We have used type classes solely for notations and not to organize abstract properties of monads such as the monad laws. Custom tactics for specific monads turned out to be much more useful. For the case of the option monad, we use a tactic that repeatedly decomposes assumptions using among the following result.

```
Lemma bind_Some {A B} (f : A → option B) (x : option A) b :
  x ≫= f = Some b ↔ ∃ a, x = Some a ∧ f a = Some b.
```

We have similar tactics that repeatedly simplify the goal using similar lemmas for the other monads such as the set and combined error state monad.

The code below in an excerpt of the Coq definition corresponding to the executable semantics $\mathsf{exec}_{\Gamma,\delta} : \mathsf{state} \to \mathcal{P}_{\mathsf{fin}}(\mathsf{state})$.

```
Definition cexec (Γ : env K) (δ : funenv K) (S : state K) : listset (state K) :=
  let 'State k ϕ m := S in
  match ϕ with
  | Expr e =>
    match maybe2 EVal e with
    | None =>
      '(E,e') ← expr_redexes e;
      match maybe_ECall_redex e' with
      | Some (Ω, f, _, _, Ωs, vs) =>
         {[ State (CFun E :: k) (Call f vs) (mem_unlock (Ω ∪ ⋃ Ωs) m) ]}
      | None =>
         let es := ehexec Γ k e' m in
         if decide (es ≡ ∅) then {[ State k (Undef (UndefExpr E e')) m ]}
         else '(e2,m2) ← es; {[ State k (Expr (subst E e2)) m2 ]}
      end
    | ...
    end
  | ...
  end.
```

The function expr_redexes decomposes an expression e into a finite set of all possible combinations (E,e') of evaluation contexts and redexes. The monadic structure is used to collect the subsequent states for each redex. The subsequent states depend on whether the redex e' is stuck, a head redex or a function call.

The code below in an excerpt of the Coq definition corresponding to the translation of statements of CH$_2$O abstract C into statements of CH$_2$O core C.

```
Definition to_stmt (τret : type K) :
    local_env K → cstmt → M (stmt K * rettype K) :=
  fix go Δl cs {struct cs} :=
  match cs with
  | CSReturn (Some ce) =>
     guard (τret ≠ voidT) with
       "return with expression in function returning void";
     '(e,τ) ← to_R_NULL τret <$> to_expr Δl ce;
     Γ ← gets to_env;
     guard (cast_typed τ τret) with "return expression has incorrect type";
     mret (ret (cast{τret} e), (true, Some τret))
  | CSWhile ce cs =>
     '(e,τ) ← to_R <$> to_expr Δl ce;
     τb ← error_of_option (maybe TBase τ)
       "conditional argument of while statement of non-base type";
     guard (τb ≠ TVoid) with
       "conditional argument of while statement of void type";
     '(s,cmσ) ← go Δl cs;
     mret (catch (loop (if{e} skip else throw 0 ;; catch s)),
          (false, cmσ.2))
  |  ...
  end.
```

9.2.3 Comparison with other Coq libraries

There are some other Coq libraries available that provide functionality similar to the CH_2O support library. We will briefly compare these libraries.

The Coq standard library [Coq15] provides abstract interfaces for finite sets and finite maps, and efficient implementations based on AVL and red-black trees. Albeit being more general, these implementations inherently require setoid equality. Furthermore, Coq's module system is used, which due to its second class nature does not allow decent overloading of notations.

The Ssreflect library by Gonthier *et al.* [GM10] provides many operations on lists, finite sets and finite functions together with a large collection of corresponding theorems. Overloaded notations and generic theory are provided through use of canonical structures (an alternative type class mechanism for Coq) and Leibniz equality is used nearly everywhere. However, the Ssreflect library is not developed for computational efficiency. Most constructions rely on generic encodings as unary natural numbers, which are general, but inherently inefficient.

The Coq extended library coq-ext-lib by Malecha [Mal15] provides abstract interfaces for finite sets, finite maps and monads based on type classes. Albeit being more abstract than the CH_2O library, only a limited number of concrete instances and a limited collection of lemmas is provided.

9.3 Overloaded typing judgments

Type classes are used to overload notations for typing judgments (we have 25 different typing judgments). We use the same approach as we have used for monads: we declare a type class and bind a notation to it. The class `Valid` is used for judgments without a type, such as $\vdash \Gamma$ and $\Gamma, \Delta \vdash m$.

```
Class Valid (E A : Type) := valid: E → A → Prop.
Notation "✓{ Γ }" := (valid Γ).
Notation "✓{ Γ }*" := (Forall (✓{Γ})).
```

We use product types to represent judgments with multiple environments such as $\Gamma, \Delta \vdash m$. The notation `✓{Γ}*` is used to lift the judgment to lists. The class `Typed` is used for judgments such as $\Gamma, \Delta \vdash v : \tau$ and $\Gamma, \Delta, \vec{\tau} \vdash e : \tau_{\mathsf{lr}}$.

```
Class Typed (E T V : Type) := typed: E → V → T → Prop.
Notation "Γ ⊢ v : τ" := (typed Γ v τ).
Notation "Γ ⊢* vs :* τs" := (Forall2 (typed Γ) vs τs).
```

9.4 Implementation-defined behavior

Type classes are used to parameterize the whole Coq development by implementation-defined parameters such as integer sizes. For example, Lemma 5.5.13 on page 83 looks like:

```
Lemma to_of_val `{EnvSpec K} Γ Δ γs v τ :
  ✓ Γ → (Γ,Δ) ⊢ v : τ → length γs = bit_size_of Γ τ →
  to_val Γ (of_val Γ γs v) = freeze true v.
```

The parameter EnvSpec K is a type class describing an implementation environment with ranks K (Definition 3.4.1 on page 48). Just as in this thesis, the type K of integer ranks is a parameter of the inductive definition of types (see Definition 3.1.1 on page 40) and is propagated through all syntax.

```
Inductive signedness := Signed | Unsigned.
Inductive int_type (K: Set) := IntType { sign: signedness; rank: K }.
```

The definition of the type class EnvSpec is based on the approach of Spitters and van der Weegen [SvdW11]. We have a separate class Env for the operations that is an implicit parameter of the whole class and all lemmas.

```
Class Env (K: Set) := {
  env_type_env :> IntEnv K;
  size_of : env K → type K → nat;
  align_of : env K → type K → nat;
  field_sizes : env K → list (type K) → list nat;
  alloc_can_fail : bool
}.
Class EnvSpec (K: Set) `{Env K} := {
  int_env_spec :>> IntEnvSpec K;
  size_of_ptr_ne_0 Γ τp : size_of Γ (τp.*) ≠ 0;
  size_of_int Γ τi : size_of Γ (intT τi) = rank_size (rank τi);
  ...
}.
```

Note that the class EnvSpec contains just logical parts that are computationally irrelevant. It will therefore not appear in the extracted OCaml code.

9.5 Partial functions

Although many operations in CH_2O are partial, we have formalized many such operations as total functions that assign an appropriate default value. We followed the approach presented in Section 4.2 where operations are combined with a *validity predicate* that describes in which case they may be used. For example, part (2) of Lemma 8.1.11 on page 149 is stated in the Coq development as follows:

```
Lemma mem_insert_union `{EnvSpec K} Γ Δ m1 m2 a1 v1 τ1 :
  ✓ Γ → ✓{Γ,Δ} m1 → m1 ⊥ m2 →
  (Γ,Δ) ⊢ a1 : TType τ1 → mem_writable Γ a1 m1 → (Γ,Δ) ⊢ v1 : τ1 →
  <[a1:=v1]{Γ}>(m1 ∪ m2) = <[a1:=v1]{Γ}>m1 ∪ m2.
```

Here, m1 ⊥ m2 is the side-condition of m1 ∪ m2, and mem_writable Γ a1 m1 the side-condition of <[a1:=v1]{Γ}>m1. Alternatives approaches include using the option monad or dependent types, but our approach proved more convenient. In particular, since most validity predicates are given by an inductive definition, various proofs could be done by induction on the structure of the validity predicate. The cases one has to consider correspond exactly to the domain of the partial function.

Admissible side-conditions, such as in the above example `<[a1:=v1]{Γ}>m1 ⊥ m2` and `mem_writable Γ a1 (m1 ∪ m2)`, do not have to be stated explicitly and follow from the side-conditions that are already there. By avoiding the need to state admissible side-conditions, we avoid a blow-up in the number of side-conditions of many lemmas. We thus reduce the proof effort needed to use such a lemma.

Our proofs of type preservation (Theorem 6.6.13 on page 123) and stability under memory refinements (Theorem 6.7.1 on page 124) give confidence that we have not forgotten any side-conditions in the operational semantics.

9.6 Automation

The proof style deployed in the CH$_2$O development combines interactive proofs with automated proofs. Proof automation is used for three main purposes:

- To automatically solve side-conditions of lemmas. For example, most lemmas related to separation algebras have side-conditions involving disjointness.
- To simplify hypotheses. For example, hypotheses involving specific monads, finite set containment, typing judgments, and some ad-hoc problems, *etc.*
- To discharge uninteresting cases in large inductive proofs. For example, in the case of inductive proofs over the small-step operational semantics, or in proofs over the various typing judgments.

In this section we describe some tactics and forms of proof automation deployed in the CH$_2$O development.

Small inversions. Coq's `inversion` tactic has two serious shortcomings on inductively defined predicates with many constructors (such as the small-step operational semantics, Definition 6.5.8 on page 118). It is slow and its way of name control for variables and hypotheses is deficient. Hence, we often used the technique of small inversions by Monin and Shi [MS13] that improves on both shortcomings.

Solving disjointness. We have used Coq's setoid machinery [Soz10] to enable rewriting using the relations $\leq_\perp$ and $\equiv_\perp$ (Definition 4.6.3 on page 59). Using this machinery, we have implemented a tactic that automatically solves entailments of the form:

$$H_0 : \perp \vec{x}_0,\ \ldots,\ H_n : \perp \vec{x}_{n-1} \quad \vdash \quad \perp \vec{x}$$

where $\vec{x}$ and $\vec{x}_i$ (for $i < n$) are arbitrary Coq expressions built from $\emptyset$, $\cup$ and $\bigcup$. This tactic works roughly as follows:

1. Simplify hypotheses using Theorem 4.6.6 on page 60.
2. Solve side-conditions by simplification using Theorem 4.6.6 and a solver for list containment (implemented by reflection).

3. Repeat these steps until no further simplification is possible.
4. Finally, solve the goal by simplification using Theorem 4.6.6 and list containment.

This tactic is not implemented using reflection, but that is something we intend to do in future work to improve its performance.

First-order logic. Many side-conditions we have encountered involve simple entailments of first-order logic such as distributing logical quantifiers combined with some propositional reasoning. Coq does not provide a solver for first-order logic apart from the `firstorder` tactic whose performance is already insufficient on small goals.

We have used Ltac to implemented an ad-hoc solver called `naive_solver`, which performs a simple breath-first search proof search. Although this tactic is inherently incomplete and suffers from some limitations, it turned out to be sufficient to solve many uninteresting side-conditions (without the need for classical axioms).

9.7 Overview of the Coq development

The Coq development, which is entirely constructive and axiom free, consists of the following parts:

Component	**Chapters**	**LOC**
Support library	Section 9.2	12 524
Types & Integers	Chapter 3	1 928
Permissions & separation algebras	Chapter 4	1 811
Memory model	Chapter 5	8 736
CH_2O core C	Chapter 6	6 979
Refinements	Section 5.8 and 6.7	6 036
CH_2O abstract C	Chapter 7	2 739
Separation logic	Chapter 8	8 467
Total		49 220

Apart from that, the translation from C source files to CH_2O abstract C involves the GNU C preprocessor, the FrontC parser, and a small layer of handwritten OCaml glue (987 LOC).

Chapter 10

Related work

There is a lot of successful work on the formalization of a wide variety of programming languages, ranging from 'clean' mathematically oriented programming languages towards real-life programming languages with many 'dirty' constructs.

Formalization has been particularly successful in the area of 'clean' mathematically oriented programming languages. A groundbreaking example is the functional programming language Standard ML whose official semantics is given in the form of typing rules and an operational semantics [MTHM97]. Initiatives such as the POPLmark challenge [ABF$^+$05] have been important to stimulate the formalization of the metatheory of programming languages using proof assistants. There have also been a lot of projects on the formalization of high-level imperative programming languages such as Java [JP04, KN06, BCG$^+$08, Loc12].

Formalization has also been successful in the area of 'dirty' low-level programming languages. For example, there are comprehensive formalizations of real-life assembly languages such as ARM [Fox03] and x86 [KWAH12]. Another interesting example is the formalization of the fragment of the C and C++ memory model for shared-memory concurrency [BOS$^+$11] where the formalization efforts have led to numerous improvements of the actual texts of the C and C++ standards.

Formalization of C is rather different from the aforementioned areas because C combines low-level aspects such as pointers and object representations with high-level aspects that allow for efficient compilation. Besides, in formalizations of higher-level programming languages, there is a significant interest in features such as algebraic data types, polymorphism, higher-order functions, module systems and object orientation. These features are absent in C. In this chapter we therefore limit ourselves to formalizations of the C language.

10.1 Formalization of the C standard

This section gives a chronological overview of formalizations of the C standard. Formalizations that only treat the C memory model are discussed in Section 10.2. Our formalization, called CH_2O, distinguishes itself from all discussed works by providing all of the following features:

- CH_2O treats a large fragment of the language.
- CH_2O is faithful to the C11 standard and is therefore compiler independent.
- CH_2O contains a formalization of the C type system and a formalized translation from abstract syntax into a principled core language.
- CH_2O has a corresponding executable and axiomatic semantics.
- CH_2O is fully formalized in a proof assistant.
- CH_2O has been used for a large number of proofs of metatheoretical results.

Figure 10.1 displays a chart of the features covered by the discussed formalizations. This chart is intended to serve both as an indication and as a summary of the more detailed description of related work on C formalizations in this section. The displayed items are not equally difficult and are subject to interpretation. We have based this chart on the comparison of formalizations by Ellison [Ell12, Figure 2.1].

Gurevich and Huggins (1993) The first attempt to give a mathematically precise semantics of C is due to Gurevich and Huggins [GH93]. Their semantics is based on the description of C in second edition of the book by Kernighan and Ritchie [KR88] and is defined in terms of the formalism of evolving algebras [Gur91]. They have not used any form of computer assistance such as a proof assistant.

Gurevich and Huggins have covered a reasonable part of the C language including some aspects of underspecification. Most notably, non-determinism in expressions is modeled using external oracles. However, as pointed out by Norrish [Nor98], they are missing interleaved evaluation orders. Consider:

```
printf("a") + (printf("b") + printf("c"));
```

Evaluation of this expression may print `bac` according to the C standard (and CH_2O), whereas Gurevich and Huggins have missed out on that behavior.

Gurevich and Huggins did not consider an executable or axiomatic semantics, and did not prove any metatheoretical properties.

Cook and Subramanian (1994) The first attempt to formalize C using a proof assistant is due to Cook and Subramanian [CS94]. Their semantics is defined in terms of a Lisp function in the proof assistant Nqthm (the predecessor of ACL2). Since their semantics is essentially an interpreter, it is by definition executable. The logic of Nqthm is used both to prove properties of simple programs and to prove some basic metatheoretical results.

The fragment of C89 treated by Cook and Subramanian includes a very limited set of types (integers, pointers to integers, arrays of integers, and functions returning an integer or void) and a limited set of expression constructs. Recursive functions are not supported, expression evaluation is assumed to go left-to-right, and implementation-defined properties (such as the size of integers) have fixed values.

C abstract syntax is translated into a smaller language. However, unlike us, they have not formalized this translation in the proof assistant itself.

	GH	CS	Nor	Pap	Ler	ER/Hat	CH$_2$O
First publication	1993	1994	1998	1998	2006	2012	2013
Standard	K&R	C89	C89	C89	C99	C11	C11
System	-	Nqthm	HOL4	Haskell	Coq	$\mathbb{K}$	Coq
Formalism	evolution algebra	executable	big/small-step	denotational	small-step	rewriting	small-step
Executable semantics	○	●	○	●	●	●	●
Axiomatic semantics	○	○	◐	○	◐	○	●
Formalized preprocessor	○	○	○	○	○	○	○
Formalized parser	○	○	○	○	●	○	○
Formalized frontend	○	○	○	○	○	●	●
Formalized compiler	○	○	○	○	●	○	○
Typing judgments	○	○	●	●	◐	○	●
Proof assistant	○	●	●	○	●	○	●
Metatheory	○	◐	●	○	●	○	●
Faithful to standard	○	○	●	●	◐	●	●
Implementation-defined	○	○	●	●	◐	●	●
Enum types	◐	○	○	●	◐	●	●
Floating point arithmetic	○	○	○	●	●	●	○
Bitfields	●	○	○	◐	◐	●	○
Struct/unions as value	○	○	●	○	○	●	●
Effective types	○	○	○	○	○	○	●
Restrictions on padding	○	○	○	○	○	◐	●
Integer conversions	◐	◐	●	●	●	●	●
Non-determinism	◐	○	●	●	●	●	●
Sequence point restriction	◐	○	●	●	○	●	●
`break`/`continue`	◐	◐	●	●	●	●	●
`goto`	◐	○	○	●	●	●	●
`switch`	◐	○	○	●	◐	●	●
`setjmp`/`longjmp`	○	○	○	○	○	●	○
`malloc`/`free`	○	○	○	○	◐	●	◐
Variadic functions	○	○	○	○	◐	●	○
External functions	○	○	○	○	●	○	○
Concurrency/atomics	○	○	○	○	◐	◐	○

● fully described ◐ partially described/part of informal translation ○ not described

Figure 10.1: Overview of formalizations of the C standard. The top rows describe properties of the formalization whereas the bottom rows describe which features of the language are covered by the formalization.

Norrish (1998) Norrish has formalized a significant fragment of the C89 standard using the proof assistant HOL4 [Nor98, Nor99]. He has considered a fragment of C with more features than earlier work, treated underspecification, and was the first to describe non-determinism and sequence points formally.

Norrish has used a big-step semantics for statements and a small-step semantics for expressions. The use of a small-step semantics was essential to describe all possible behaviors due to non-deterministic expression evaluation, but the combination proved less fruitful. In the conclusion of his PhD thesis [Nor98, Section 7.1], Norrish states that he regrets having used a big-step semantics for statements. Our treatment of non-determinism and sequence points is inspired by Norrish's, see Section 6.4.

Apart from the operational semantics, Norrish has also formalized the type system of C and proven type preservation. Norrish has omitted some features of C that can be handled by translation (such as static local variables, typedefs and enum types) but has explicitly treated other features that we handle by translation (such as l-value conversion). He has not formalized the translation that simplifies these features.

Norrish has proven the validity of Hoare rules for statements and confluence results for sequence point free expressions in order to facilitate reasoning about concrete C programs. Although his confluence results do not hold in our formalization, variants of his proofs may also be useful for more efficient symbolic evaluation of expressions. He has left reasoning about arbitrary C expressions as an open problem.

Contrary to our work, Norrish has used an unstructured memory model based on sequences of bytes. Since he has considered the C89 standard in which effective types (and similar notions) were not introduced yet, his choice is appropriate. For C99 and beyond, a more detailed memory model like ours is needed, see also Section 2.5 and Defect Report #260 and #451 [ISO].

Another interesting difference is that Norrish represents abstract values (integers, pointers and structs) as sequences of bytes instead of mathematical values. Due to this, padding bytes retain their value while structs are copied. This is not faithful to the C99 standard and beyond.

Papaspyrou (1998) Papaspyrou has defined a denotational semantics for a significant fragment of the C89 standard in the form of a Haskell program [Pap98]. Monads and monad transformers are used to model different aspects of computations, including more difficult aspects of C such as non-determinism and sequence points.

Papaspyrou's goal is similar to Norrish's, namely to describe the C89 as accurately as possible. However, his approach is entirely different from Norrish's. Most notably, he uses a denotational semantics instead of an operational semantics and considers a larger fragment of the C language. For example, his fragment includes floats, unions, `goto`s and `switch` statements, whereas Norrish has omitted these features. On the other hand, Norrish has used a proof assistant to define his semantics and is therefore able to formally prove metatheoretical results.

Papaspyrou has described the C semantics using monads. Since all parts of his semantics are defined in Haskell, the semantics is by definition executable. Similar to our interpreter (Chapter 7) it can be used to compute the set of all possible behaviors of a program. However, as observed by Ellison [Ell12], Papaspyrou's interpreter is of little use due to performance problems.

Leroy *et al.* (2006) Leroy *et al.* have formalized a significant part of C using the Coq proof assistant [Ler06, Ler09b]. Their part of C, called CompCert C, covers most major features of C and can be compiled into assembly (PowerPC, ARM and x86) using a compiler written in Coq. Their compiler, called CompCert, has been proven correct with respect to the CompCert C and assembly semantics.

The goal of CompCert is essentially different from CH_2O's. What can be proven with respect to the CompCert semantics does not have to hold for *any* C11 compiler, it just has to hold for the CompCert compiler. CompCert is therefore in its semantics allowed to restrict implementation-defined behaviors to be very specific (for example, it uses 32-bit `int`s since it targets only 32-bit computing architectures) and allowed to give a defined semantics to various undefined behaviors of C11 (such as sequence point violations, violations of effective types, and certain uses of dangling pointers). Let us consider a simple example:

```
int *p;
{ int x = 10; p = &x; }
printf("%d\n", *p);
```

This code has undefined behavior according to the C11 standard as well as CH_2O because the value of the dangling pointer `p` is used. However, in CompCert this code is guaranteed to print 10 because all local variables have function scope. As discussed in Section 2.3, CompCert is allowed to do so. Other differences between CompCert, CH_2O and C11 are discussed in Section 2.3.

CompCert uses a small-step semantics for the source language CompCert C, the intermediate languages used in the compiler, and the targeted assembly languages (PowerPC, ARM and x86). Earlier versions of CompCert have used a big-step operational semantics with a coinductive counterpart for non-terminating programs [LG09]. However, the big-step semantics has been replaced by a small-step semantics to describe non-local flow such as `goto` statements.

Apart from the operational semantics, CompCert has also formalized typing judgments and a proof of type preservation. Compared to the typing judgments of CH_2O there are some differences:

- Our typing judgments are stronger. For example, we guarantee that all pointers refer to objects that actually exist in memory, that the memory is well-formed, and that `goto`s have a corresponding label.
- Our semantics enjoys a progress property.
- Our translation from abstract syntax is written in Coq and is proven to be type sound in Coq. CompCert's translation from abstract syntax mainly consists of unverified OCaml code. It uses some verified helper functions that are obtained by extraction from Coq.

The parser of CompCert is formally verified [JPL12]. That means, it has a formal definition of the C grammar and its parser has been proven sound and complete with respect to that definition of the C grammar.

CompCert has significantly influenced this thesis. Most notably, the foundations of our memory model are based on the CompCert memory model [LB08, LABS12], and our notion of memory refinements (Section 5.8) is an adaptation of CompCert's notion of memory injections. But whereas CompCert uses an untyped memory model that consists of arrays of bytes, our memory model consists of well-typed trees in order to faithfully describe the C11 standard.

CompCert has been used for numerous applications. Appel *et al.* have developed a separation logic for CompCert [App14], which for example has been used to verify an implementation of SHA [App15]. Jourdan *et al.* have developed a formally verified static analyzer for CompCert's intermediate language C$^\sharp$-minor [JLB$^+$15]. Shao *et al.* have verified a microkernel operating system against a variant of the CompCert semantics [GVF$^+$11]. Furthermore, various extensions of both the CompCert compiler as well as the CompCert semantics have been developed. For example:

- Dargaye has developed a compiler from a subset of ML to an intermediate language of CompCert [Dar09].
- Barthe *et al.* have developed a Static Single Assignment (SSA) based middle-end for CompCert [BDP12].
- Sevcík *et al.* have extended CompCert with shared-memory concurrency and atomics [SVN$^+$13]. Their extension is based on their x86-TSO model [SSO$^+$10] and is thus called CompCert TSO. They have extended many intermediate languages and proofs of compilation phases.
- Carbonneaux *et al.* [CHRS14] have developed an extension of CompCert, called Quantitative CompCert, that takes bounds on stack usage into account. On top of that, they provide a Hoare logic to establish bounds on stack usage, and a verified stack usage analyzer.
- Beringer *et al.* have developed an extension of CompCert, called Compositional CompCert, for separate compilation [BSDA14, SBCA15]. Separate compilation is the process of compiling individual modules of a program separately while preserving the correctness of the whole program. They have proposed a semantical approach for modeling the interaction between different modules, which themselves may be written in different languages. They have successfully applied their approach to extend the CompCert semantics and compiler proofs with support for separate compilation.
- Kang and Hur have also developed a version of CompCert suitable for separate compilation [KH15]. Contrary to Beringer *et al.*, they use a syntactical method instead of a semantical one.

Ellison and Roşu (2012) Ellison and Roşu [ER12b, Ell12] have developed an executable semantics of the C11 standard using the $\mathbb{K}$-framework[1]. Their semantics is very comprehensive and describes all features of a freestanding C implementation [ISO12, 4p6] including some parts of the standard library. It furthermore has

[1] This work has been superseded by Hathhorn *et al.* [HER15], which is described below. Table 10.1 therefore only includes Hathhorn *et al.*.

been thoroughly tested against test suites (such as the GCC torture test suite), and has been used as an oracle for compiler testing [RCC+12].

Ellison and Roşu support more C features than we do, but they do not have infrastructure for formal proofs, and thus have not established any metatheoretical properties about their semantics. Their semantics, despite being written in a formal framework, should more be seen as a debugger, a state space search tool, or possibly, as a model checker. It is unlikely to be of practical use in proof assistants because it is defined on top of a large C abstract syntax and uses a rather ad-hoc execution state that contains over 90 components.

Similar to our work, Ellison and Roşu's goal is to *exactly* describe the C11 standard. However, for some programs their semantics is less precise than ours, which is mainly caused by their memory model, which is less principled than ours. Their memory model is based on CompCert's: it is essentially a finite map of objects consisting of unstructured arrays of bytes.

Ellison and Roşu provide a comprehensive treatment of the operational semantics but do not treat the type system. Type checking is performed during execution only whenever necessary. Also, some parts of the static semantics such as name binding are handled dynamically. This has some strange consequences, for example:

```
int main() {
  if(0) { return x; } else { return 12; }
}
```

The C11 standard requires compilers to produce a diagnostic in case an undeclared variable is used [ISO12, 5.1.1.3p1, 6.5.1p2], but Ellison and Roşu require this program to return 12. In fact, it seems they allow unreachable parts of the program to contain any syntactically valid blob of code. The above program is statically rejected by the type system of CompCert and CH_2O.

Hathhorn *et al.* (2015) Hathhorn *et al.* [HER15] have extended the work of Ellison and Roşu to handle more underspecification of C11. Most importantly, the memory model has been extended, support for the type qualifiers `const`, `restrict` and `volatile` has been added, and the static semantics (which includes linking and type checking) is now handled as part of the formal semantics. This extension increased the size of their semantics, in terms of lines of code, by a factor of two.

Hathhorn *et al.* have extended the original memory model (which was based on CompCert's) with decorations to handle effective types, restrictions on padding and the `restrict` qualifier. Effective types are modeled by a map that associates a type to each object. Their approach is less fine-grained than ours and is unable to account for active variants of unions. It thus does not assign undefined behavior to important violations of effective types and in turn does not allow compilers to perform optimizations based on type-based alias analysis. For example:

```
// Undefined behavior in case f is called with aliased
// pointers due to effective types
int f(short *p, int *q) { *p = 10; *q = 11; return *p; }
int main() {
  union { short x; int y; } u = { .y = 0 };
  return f(&u.x, &u.y);
}
```

The above program has undefined behavior due to a violation of effective types. This is captured by our tree based memory model, but Hathhorn *et al.* require the program to return the value 11. When compiled with GCC or Clang with optimization level `-O2`, the compiled program returns the value 10.

Hathhorn *et al.* handle restrictions on padding bytes in the case of unions, but not in the case of structs. For example, the following program returns the value 1 according to their semantics, whereas it has unspecified behavior according to the C11 standard [ISO12, 6.2.6.1p6] (see also Section 2.5.3):

```
struct S { char a; int b; } s;
((unsigned char*)(&s))[1] = 1;
s.a = 10; // Makes the padding bytes of 's' indeterminate
return ((unsigned char*)(&s))[1];
```

The restrictions on paddings bytes are implicit in our memory model based on structured trees, and thus handled correctly. The above examples provide evidence that a structured approach, especially combined with metatheoretical results, is more reliable than depending on ad-hoc decorations.

Hathhorn *et al.* have furthermore extended their original semantics with a formalized translation phase that performs type checking and translation of static constructs (such as initializers and array sizes). The example in the paragraph Ellison and Roşu (2012) therefore no longer applies to this version.

An interesting feature of this translation phase is support for linking of multiple `.c` source files, whereas CH_2O only supports individual source files. The approach of Hathhorn *et al.* to linking is syntactical and thus essentially different from the semantical approach of Beringer *et al.* in the context of CompCert [BSDA14, SBCA15]. Ideally, one would like to consider a notion of syntactic and semantic linking that are proven to correspond with each other.

Memarian *et al.* (2015) Cerberus is an ongoing project by Memarian *et al.*, described in the draft paper [MNM$^+$15], that aims at developing a semantics of a *de facto version of C*[2]. Instead of formalizing C as described by the official ISO C11 standard, their formalization is based on the way C is used in practice. Their goal is thus different from ours.

As a first step to investigate how C is used in practice, Memarian and Sewell have undertaken a web survey [MS15]. They have received around 300 responses, including many responses by systems programmers and compiler writers. The survey consisted of 15 questions and focused on issues related to pointers and memory, similar to those that we have discussed in Section 2.5.

The responses to the survey display a wide diversity of answers. Although some constructs considered in certain questions are clearly undefined according to the C11

[2] Since the paper is an unpublished draft, we have not included it in Table 10.1.

standard (and thus also in CH_2O), many respondents note that widely used code relies on these constructs, whereas others note that these may be miscompiled. Cerberus intends to provide switchable options for different behaviors of such constructs.

A draft paper by Memarian *et al.* [MNM$^+$15] describes an initial version of the Cerberus formalization. Similar to out work, actual C programs are translated into a core language. This translation performs type checking. Similar to CH_2O core C, the Cerberus core language has an executable semantics, but is also very different from CH_2O core C. For example:

- The Cerberus core language is an imperative strongly typed functional language with first-order recursive functions. It is thus further away from C than both CH_2O core C and CompCert C.
- All C-style loops are modeled using labeled statements and `goto`s. This reduces the size of the core language, but at the cost of having to deal with freshness conditions of labels for already simple loops.
- It has explicit constructs to deal with the lifetime of objects. These constructs are for example used when a goto leaves a block scope containing local variables. CH_2O core C implicitly deals with the lifetime of objects.
- It has explicit constructs for undefined behavior. For example, division of `x` by `y` is encoded as `if y = 0 then undef(Division by zero) else x / y`. CH_2O core C has a similar expression for undefined behavior, but it is only used to capture undefined behavior due to control reaching the end of a non-`void` function. Other undefined behavior is implicit in CH_2O core C.
- Whereas CH_2O does not consider concurrency at all, Cerberus uses a concurrent memory model based on the work by Batty *et al.* [BOS$^+$11]. Non-determinism in expressions (as well as undefined behavior due to sequence point violations) is modeled using concurrency primitives.

The current version of the Cerberus formalization does not deal with byte representations, unions, and other issues related to the memory and pointers. This is left for future work based on the results of the survey.

10.2 Formalization of the C memory model

The memory model is arguably the most challenging part of a formalized C semantics. This is mainly caused by the very subtle interaction between low-level and high-level memory access. In this thesis we have proposed a structured memory model based on well-typed trees to close the gap between these two levels.

The idea of using a memory model based on trees instead of arrays of plain bits, and the idea of using pointers based on paths instead of offsets, has already been used for object oriented languages. It goes back at least to Rossie and Friedman [RF95], and has been used by Ramananandro *et al.* [RDRL11] for C++. Furthermore, many researchers have considered connections between unstructured and structured views of data in C [TKN07, CMTS09, AS14, GLAK14] in the context of program logics.

However, a memory model that combines an abstract tree based structure with low-level object representations in terms of bytes has not been explored before. In this section we will describe other formalizations of the C memory model.

Leroy and Blazy (2008) The CompCert memory model is used by all languages (from C until assembly) of the CompCert compiler [LB08, LABS12]. The CompCert memory is a finite partial function from object identifiers to objects. Each local, global and static variable, and invocation of `malloc` is associated with a unique object identifier of a separate object in memory. This approach is well-suited for reasoning about memory transformations because it separates unrelated objects. We have used the same approach in CH_2O.

In the first version of the CompCert memory model [LB08], objects were represented as arrays of type-annotated fragments of base values. Examples of bytes are thus “the 2nd byte of the short 13” or “the 3rd byte of the pointer (o, i)”. Pointers were represented as pairs (o, i) where o is an object identifier and i the byte offset into the object o.

Since bytes are annotated with types and could only be retrieved from memory using an expression of matching type, effective types on the level of base types are implicitly described. However, this does not match the C11 standard. For example, Leroy and Blazy do assign defined behavior to the following program:

```
struct S1 { int x; };
struct S2 { int y; };
int f(struct S1 *p, struct S2 *q) {
  p->x = 10;
  q->y = 11;
  return p->x;
}
int main() {
  union U { struct S1 s1; struct S2 s2; } u;
  printf("%d\n", f(&u.s1, &u.s2));
}
```

This code strongly resembles example [ISO12, 6.5.2.3p9] from the C11 standard, which is stated to have undefined behavior[3]. GCC and Clang optimize this code to print 10, which differs from the value assigned by Leroy and Blazy’s memory model.

Apart from assigning too much defined behavior, Leroy and Blazy’s treatment of effective types also prohibits any form of “bit twiddling”.

Leroy and Blazy have introduced the notion of memory injections in [LB08]. This notion allows one to reason about memory transformations in an elegant way. Our notion of memory refinements (Section 5.8) generalize the approach of Leroy and Blazy to a tree based memory model.

[3] We have modified the example from the standard slightly in order to trigger optimizations by GCC and Clang.

Leroy *et al.* (2012) The second version of CompCert memory model [LABS12] is entirely untyped and is extended with permissions. Symbolic bytes are only used for pointer values and indeterminate storage, whereas integer and floating point values are represented as numerical bytes (integers between 0 and $2^8 - 1$).

We have extended this approach by analogy to bit-representations, representing indeterminate storage and pointer values using symbolic bits, and integer values using concrete bits. This choice is detailed in Section 5.3.

Besson *et al.* (2014) Besson *et al.* have proposed an extension of the CompCert memory model that assigns a defined semantics to operations that rely on the numerical values of uninitialized memory and pointers [BBW14].

Objects in their memory model consist of lazily evaluated values described by symbolic expressions. These symbolic expressions are used to delay the evaluation of operations on uninitialized memory and pointer values. Only when a concrete value is needed (for example in case of the controlling expression of an `if-else`, `for`, or `while` statement), the symbolic expression is normalized. Consider:

```
int x, *p = &x;
int y = ((unsigned char*)p)[1] | 1;
// y has symbolic value "2nd pointer byte of p" | 1
if (y & 1) printf("one\n"); // unique normalization -> OK
if (y & 2) printf("two\n"); // no unique normalization -> bad
```

The value of `((unsigned char*)p)[1] | 1` is not evaluated eagerly. Instead, the assignment to `y` stores a symbolic expression denoting this value in memory. During the execution of the first `if` statement, the actual value of `y & 1` is needed. In this case `y & 1` has the value 1 for any possible numerical value of `((unsigned char*)p)[1]`. As a result, the string `one` is printed.

The semantics of Besson *et al.* is deterministic by definition. Normalization of symbolic expressions has defined behavior if and only if the expression can be normalized to a unique value under any choice of numeral values for pointer representations and uninitialized storage. In the second `if` statement this is not the case.

The approach of Besson *et al.* gives a semantics to some programming techniques that rely on the numerical representations of pointers and uninitialized memory. For example, it gives an appropriate semantics to pointer tagging in which unused bits of a pointer representation are used to store additional information.

However, as already observed by Kang *et al.* [KHM$^+$15], Besson *et al.* do not give a semantics to many other useful cases. For example, printing the object representation of a struct, or computing the hash of a pointer value, is inherently non-deterministic. The approach of Besson *et al.* assigns undefined behavior to these use cases.

The goal of Besson *et al.* is inherently different from ours. Our goal is to describe the C11 standard faithfully whereas Besson *et al.* focus on *de facto* versions of C. They intentionally assign defined behavior to many constructs involving uninitialized memory that are clearly undefined according to the C11 standard, but that are nonetheless faithfully compiled by specific compilers.

Kang *et al.* (2015) Kang *et al.* [KHM+15] have proposed a memory model that gives a semantics to pointer to integer casts. Their memory model uses a combination of numerical and symbolic representations of pointer values (whereas CompCert and CH_2O always represent pointer values symbolically). Initially each pointer is represented symbolically, but whenever the numerical representation of a pointer is needed (due to a pointer to integer cast), it is non-deterministically *realized.*

The memory model of Kang *et al.* gives a semantics to pointer to integer casts while allowing common compiler optimizations that are invalid in a naive low-level memory model. They provide the following motivating example:

```c
void g(void) { ... }
int f(void) {
  int a = 0;
  g();
  return a;
}
```

In a concrete memory model, there is the possibility that the function `g` is able to *guess* the numerical representation of `&a`, and thereby access or even modify `a`. This is undesirable, because it prevents the widely used optimization of constant propagation, which optimizes the variable `a` out.

In the CompCert and CH_2O memory model, where pointers are represented symbolically, it is guaranteed that `f` has exclusive control over `a`. Since `&a` has not been leaked, `g` can impossibly access `a`. In the memory model of Kang *et al.* a pointer will only be given a numerical representation when it is cast to an integer. In the above code, no such casts appear, and `g` cannot access `a`.

The goal of Kang *et al.* is to give a unambiguous mathematical model for pointer to integer casts, but not necessarily to comply with C11 or existing compilers. Although we think that their model is a reasonable choice, it is unclear whether it is faithful to the C11 standard in the context of Defect Report #260 [ISO]. Consider:

```c
int x = 0, *p = 0;
for (uintptr_t i = 0; ; i++) {
  if (i == (uintptr_t)&x) {
    p = (int*)i;
    break;
  }
}
*p = 15;
printf("%d\n", x);
```

Here we loop through the range of integers of type `uintptr_t` until we have found the integer representation `i` of `&x`, which we then assign to the pointer `p`.

When compiled with `gcc -O2` (version 4.9.2), the generated assembly no longer contains a loop, and the pointers `p` and `&x` are assumed not to alias. As a result, the program prints the old value of `x`, namely `0`. In the memory model of Kang *et al.* the pointer obtained via the cast `(int*)i` is exactly the same as `&x`. In their model the program thus has defined behavior and is required to print `15`.

We have reported this issue to the GCC bug tracker[4]. However it unclear whether the GCC developers consider this a bug or not. Some developers seem to believe that this program has undefined behavior and that GCC's optimizations are thus justified. Note that the cast `(intptr_t)&x` is already forbidden by the type system of CH_2O.

10.3 Program logics for C

This section compares our axiomatic semantics with other program logics for C. Our axiomatic semantics distinguishes itself from all existing work by taking all forms of underspecification of C seriously. In particular, we correctly treat non-deterministic expressions evaluation in the presence of side-effects and block scope local variables in the presence of non-local control flow.

The idea of using a separate Hoare judgment for statements and expressions is not new. Von Oheimb [Ohe01, vON02] has used a Hoare judgment for expressions in the context of an axiomatic semantics for Java in Isabelle. The shape of his judgments for expressions is similar to ours, but his inference rules are not. Since Von Oheimb considered Java, he was able to rely on left-to-right expression evaluation, which one cannot do in C. Hoare judgments whose postcondition is a function from values to assertions are also used in Hoare and separation logics for type theories and functional languages, see for example [Kri11].

Black and Windley (1996) Black and Windley have developed an axiomatic semantics for C and have used it to verify a simple web server using the HOL proof assistant [BW96]. They have defined inference rules to factor out side-effects of expressions by translating these into semantically equivalent expressions. Their axiomatic semantics supports a limited set of expression constructs and soundness has not been proven with respect to an operational semantics.

Appel and Blazy (2007) Appel and Blazy have developed a separation logic for a subset of C [AB07]. Most notably, their separation logic supports **`return`**, `catch` and `throw` statements. Our treatment of these constructs is based on theirs.

The axiomatic semantics of Appel and Blazy is a shallow embedding on top of a small-step operational semantics for the intermediate language Cminor of CompCert. In Cminor, expressions have been determinized and side-effects have been pulled out. Their axiomatic semantics is thus limited to verification of programs compiled with CompCert, and will not provide reliable guarantees if the program is compiled using another compiler.

Appel *et al.* (2011) The Verified Software Toolchain (VST) by Appel *et al.* provides a higher-order separation logic for Verifiable C, which is a variant of CompCert's intermediate language Clight [App11, App14]. The VST has for example been used to verify an implementation of SHA [App15].

[4] See `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=65752` .

The VST is intended to be used together with the CompCert compiler. It gives very strong guarantees when done so. The soundness proof of the VST in conjunction with the correctness proof of the CompCert compiler ensure that the proven properties also hold for the generated assembly.

In case the verified program is compiled with a compiler different from CompCert, the trust in the program is still increased, but no full guarantees can be given. That is caused by the fact that CompCert's intermediate language Clight uses a specific evaluation order and assigns defined behavior to many undefined behaviors of the C11 standard. For example, Clight assigns defined behavior to violations of effective types, sequence point violations, and overflow of signed integers. The VST inherits these defined behaviors from CompCert and allows one to use them in proofs.

The VST provides a type system based on static analysis to reduce the number of side-conditions when applying separation logic rules [DA13]. Contrary to the $\mathrm{CH_2O}$ type system, it gives stronger guarantees about run-time properties. For example, it keeps track of whether variables are known to be initialized. It would be interesting to investigate whether a similar type system can be applied to the $\mathrm{CH_2O}$ separation logic. In particular, whether it can be extended to keep track of non-aliasing properties related to effective types and multiple side-effects in expressions.

Since the VST is tightly linked to CompCert, it uses CompCert's coarse permission system on the level of the operational semantics. Stewart and Appel [App14, Chapter 42] have introduced a way to use a more fine grained permission system at the level of the separation logic without having to modify the Clight operational semantics. Their approach shows its merits when used for concurrency, in which case the memory model contains *ghost* data related to the conditions of locks [Hob08, HAN08].

Herms (2013) Herms has formalized a verification condition generation in the style of Why3 and Jessie in Coq [Her13]. His verification condition generator can be used as a standalone tool via Coq's extraction mechanism. Herms has proven soundness with respect to an intermediate language of CompCert.

Affeldt *et al.* (2013) Affeldt *et al.* have formalized an operational semantics and separation logic for a small fragment of C using the Coq proof assistant [AM13, AS14]. They have applied their separation logic to verify a part of an implementation of the Transport Layer Security (TLS) protocol.

The interests of Affeldt *et al.* lie in the verification of particular algorithms. For that reason, they have considered essentially a `while` language with side-effect free C-like expressions. Their language supports undefined behavior of integer arithmetic and casts, as well as pointers, alignment, and struct types.

Objects in their memory model consist of unstructured sequences of bytes, but to provide an abstract view of values in their separation logic, they make use of a special singleton connective. Their singleton connective has similarities with ours.

Their expressions have been represented in a dependently typed fashion. That means, expressions are indexed by types, and that way, untyped expressions cannot be constructed by definition. This approach is appealing from a mathematical perspective, but due to the absence of proof irrelevance and dependent pattern matching in Coq, we are not convinced it scales to a large part of a language like C.

Most problematically, in our formalization many typing judgments, including the expression judgment $\Gamma, \Delta, \vec{\tau} \vdash e : \tau_{\mathsf{lr}}$, depend on a memory typing environment Δ, which describes the layout of the memory (see Definition 5.2.2 on page 64). When one tries to encode our typing judgments into the syntax using dependent types, the Coq type of each expression will depend on the memory typing environment Δ. Since memory typing environment evolve during program execution, this thus also means that the Coq types of expressions change during program execution.

Greenaway *et al.* (2014) The AutoCorres tool by Greenway *et al.* provides an approach to C verification that is rather different from the aforementioned related work [GLAK14]. AutoCorres translates C source code into monadic definitions inside the Isabelle proof assistant, which can then be verified using separation logic or any other form of verification condition generation. It has been used in the L4.verified project by Klein *et al.* [K+09] to verify a microkernel operating system.

The translation of C sources into monadic definitions uses Norrish's C to Isabelle parser [TKN07] to translate C sources into an intermediate language. Various verified translations are performed to translate programs in this intermediate language into a monadic shallow embedding. Their approach allows one to reason about C programs while abstracting from some low-level details.

The fragment of C supported by Greenway *et al.* is intentionally kept small. For instance, pointers to local variables, function pointers, union types and expressions with side-effects are not supported. AutoCorres takes undefined behavior of integer overflow seriously, but makes compiler and architecture specific assumptions about the memory model. These assumptions are not faithful with respect to the C11 standard. However, they have used translation validation to establish correctness of the assembly produced by GCC [SMK13]. That means, for concrete programs they have an automatic means to establish that the semantics of the assembly matches the semantics of the C sources in the proof assistant.

The AutoCorres approach is interesting and we see no fundamental reason why it cannot be incorporated with a memory model like ours that is faithful to the C11 standard. Given that each local, global and static variable, and result of `malloc` is implicitly separated in our memory model, it may even provide some simplifications. Handling non-deterministic expression evaluation in a monadic style in a way that it allows for convenient reasoning seems more challenging.

Chapter 11

Conclusion and future work

In this thesis we have given a formal description of a significant part of the C11 standard. This formal description embodies the type system, an operational, executable, and axiomatic semantics, and a comprehensive collection of metatheoretical results. All of these results have been formalized using the Coq proof assistant. Our results confirm that proof assistants are well-suited to describe and reason about non-trivial real-life programming languages.

"Keep the language small and simple" is one of the design choices from the rationale of the C standard [ISO03]. Although C99 and C11 have kept C relatively small in terms of number of features (especially compared to languages such as C++), C is clearly not kept simple. The standard committee's desire to exploit C as a low-level systems programming language, *as well as* a high-level language that allows effective optimizations, complicates the semantics of features that are expected to be simple. This complication is witnessed by the many examples of subtle corner cases that we have presented. These examples expose discrepancies between the standard, the committee's judgment in defect reports, compilers, and what many programmers expect. These discrepancies are not only of academic interest since widely used C compilers optimize programs in ways that may be unexpected to the naive user.

The CH_2O semantics is formulated in a mathematically precise way and thus does not suffer from any ambiguities. It should provide a benefit to compiler writers and programmers, who will get the means to establish how the standard needs to be understood without having to deal with ambiguities of prose style natural language. Our executable semantics even allows one to compute all behaviors of a given program without the need to understand the mathematical formalism.

It would have been impossible to achieve the presented results without the support of a proof assistant. Prose style natural language, as used in conventional standard texts, is prone to errors, but informal mathematics suffers from that problem as well, especially when applied to real life programming languages such as C. The ability of proof assistants to formally state definitions and to reason about these definitions in a unified framework is essential to describe real-life programming languages. Without the support of a proof assistant to reason about one's definitions, one will inevitably forget about corner cases.

This chapter gives an overview of possible directions for future work, and describes general prospects on formal C verification. In particular, we argue how proof assistants and the results in this thesis can improve the current state of the art in program verification and programming language development.

11.1 Formalization and standardization

We believe that proof assistants are powerful tools that should be part of the development of programming language specifications. This is not to say that programming languages standards should be replaced entirely by definitions in a proof assistant, but it is to say that prose style definitions and formal definitions should be developed hand in hand.

11.1.1 Feature interaction

Feature interaction is the biggest source of problems in the standardization process of a programming language. When using a prose style definition, features are introduced by adding various paragraphs to the text while modifying other paragraphs. Although such extensions are typically reviewed extensively, there is no rigorous way to establish that an extension does not introduce unwanted feature interaction and does not introduce other kinds of inconsistencies. Formal definitions in a proof assistant do not suffer from these problems.

First of all, in a formal definition one has to be explicit about the data structures that describe the program state. In order to extend the language with a new feature, one has to explicitly reconsider the data structures involved, and modify definitions accordingly. This level of explicitness already provides a stringent sanity check.

In prose style definitions such as the C standard, already the description of the memory state is left implicit. This is especially troublesome in the case of extensions of the language such as effective types [ISO12, 6.5p6-7] and the standard committee's judgment in Defect Reports #260 and #451 [ISO] that require major modifications to the traditional K&R and C90 memory model. Such changes have never been made explicit in the standard text, and it is therefore very difficult to establish how these interact with already existing features.

The real advantage of a proof assistant is that it enables one to prove metatheoretical results about the language specification. Relatively simple properties such as type preservation already provide a decent sanity check. But most importantly, by developing different versions of the semantics (CH$_2$O has an operational, executable and axiomatic semantics) one considers the language from different perspectives and problems are easily spot. A crucial sanity check is to ensure that the different versions of the semantics correspond and keep on corresponding while new features are added. Without a proof assistant it would be infeasible to do so.

11.1.2 Executable semantics

An executable semantics is a crucial artifact to communicate the formal definition of a programming language semantics to a user. It allows one to evaluate the semantics without having to understand the definitions to their full extent. Proof assistants provide a means to implement an executable semantics that is guaranteed to correspond to the operational semantics.

11.1.3 Improve the C standard

The goal of this thesis was to take the C11 standard and existing compilers as given, and thus to ensure that whenever one proves something about a given program with respect to our semantics, the proven property holds when the program has been compiled with *any* C11 compliant compiler. So, we have assigned undefined behavior in case the C11 standard is unclear, or in case a construct is compiled in an unexpected way by a widely used compiler such as GCC and Clang.

It would be interesting to investigate whether our semantics can be used to help the standard committee to improve future versions of the standard. For example, whether it could help to improve the standard's prose description of effective types. As indicated in Section 2.5.5, the standard's description is not only ambiguous, but also does not cover its intent to make type-based alias analysis possible. A formal semantics such as CH_2O is unambiguous and allows one to express intended consequences formally. We have formally proven soundness of an abstract version of type-based alias analysis with respect to our memory model (Theorem 5.7.2 on page 87).

It would be interesting to investigate a formal semantics that addresses the issues regarding integer representations of indeterminate memory and pointers as explained in Sections 2.6.1 and 2.6.2. Given that the standard committee seems not to know how to clarify the prose text related to these issues (see their discussion in Defect Report #451 [ISO]), formalization may provide prospects for improvements. The recent work by Kang *et al.* [KHM+15] provides a step into that direction, but in the context of a much smaller language than the entire C language.

11.2 Future work

11.2.1 Features of the language

An obvious direction for future work is to extend CH_2O with additional features. We give an overview of some features of C11 that are absent in CH_2O.

- **Formalized parser and preprocessor.** We currently use the FrontC parser, which is written in OCaml. It would be interesting to use the C parser by Jourdan *et al.* [JPL12], which has been implemented and verified with respect to the C grammar in Coq. The use of a formalized parser also opens the door to a formal description of the C preprocessor.
- **Floating point arithmetic.** Representations of floating point numbers and the behaviors of floating point arithmetic are subject to a considerable amount of implementation-defined behavior [ISO12, 5.2.4.2.2].

  First of all, one could restrict to IEEE-754 floating point arithmetic, which has a clear specification [IEE08] and a comprehensive formalization in Coq [BM11].

Boldo *et al.* have taken this approach in the context of CompCert [BJLM13] and we see no fundamental problems applying it to CH_2O as well.

Alternatively, one could consider formalizing all implementation-defined aspects of the description of floating arithmetic in the C11 standard.

- **Bitfields.** Bitfields are fields of struct types that occupy individual bits [ISO12, 6.7.2.1p9]. We do not foresee fundamental problems adding bitfields to CH_2O as bits already constitute the smallest unit of storage in our memory model.
- **Untyped malloc.** CH_2O supports dynamic memory allocation via an operator $\mathsf{alloc}_\tau\ e$ close to C++'s `new` operator. The $\mathsf{alloc}_\tau\ e$ operator yields a $\tau*$ pointer to storage for a τ-array of length e. This is different from C's `malloc` function that yields a **`void*`** pointer to storage of unknown type [ISO12, 7.22.3.4].

  Dynamic memory allocation via the untyped `malloc` function is closely related to unions and effective types. Only when dynamically allocated storage is actually used, will it receive an effective type. We expect one could treat `malloc`ed objects as unions that range over all possible types that fit.
- **Variadic functions.** The C11 standard supports functions with a variable number of arguments of varying types [ISO12, 7.16]. The prototypical example is the `printf` function [ISO12, 7.21.6].

  Giving a correct and elegant formal treatment of variadic functions that is both faithful to the C11 standard and allows for formal proofs seems difficult. The operations provided by C11 to access values of variadic arguments are low-level, not type safe, and constrained by tricky side-conditions.
- **Register storage class.** The register storage class is a hint that recommends the compiler to place a variable in a register.

  The register storage class obviously affects the static semantics which has to ensure that pointers to register variables cannot be created. More surprisingly, it also affects the operational semantics. Register variables are subject to stricter rules when left uninitialized [ISO12, 6.3.2.1p2]. These rules also apply in case a given variable has not been explicitly declared with register storage class, but *could have been* declared so.

  The register storage class is currently not supported by CH_2O, and we therefore miss out on some undefined behaviors related to variables that could have been declared with it. We do not not foresee fundamental problems adding it.
- **Type qualifiers.** The **`const`**, **`restrict`** and **`volatile`** type qualifiers [ISO12, 6.7.3] are currently unsupported by CH_2O.

  The **`const`** qualifier can be applied to any type to indicate that the values do not change. Given that C does not enjoy const safety[1], **`const`** is not just part of the static type system but is also part of the memory model and operational semantics. Since the CH_2O permission system already supports **`const`** qualified permissions, we do not foresee fundamental problems.

[1] For example, the standard library function `strstr` takes a **`const`** qualified string pointer and returns a non-**`const`** qualified pointer within the given string.

  The `restrict` qualifier can be applied to any pointer type to express that the pointers do not alias. Since the description in the C11 standard [ISO12, 6.7.3.1] is ambiguous (most notably, it is unclear how it interacts with nested pointers and data types), formalization may provide prospects for clarification.

  The **`volatile`** qualifier can be applied to any type to indicate that its value may be changed by an external process. It is meant to prevent compilers from optimizing away data accesses or reordering these [ISO12, footnote 134]. Volatile accesses should thus be considered as a form of I/O.

- **External functions and I/O.** An important direction for future work is to consider C programs that interact with an outside world.

  Whereas the CH_2O small-step operational semantics is just a relation between states, CompCert uses a labeled transition system. The labels represent interactions with the outside world such as calls to external functions (printing, file access, *etc.*) and accesses to **`volatile`** qualified storage. CompCert's treatment of external functions suffers from some limitations, most notably, it requires each external function to terminate.

  In order to remedy the shortcomings of CompCert's treatment of external functions, Beringer *et al.* [BSDA14, SBCA15] have proposed a general framework in which multiple operational semantics can communicate with each other via external functions. They have applied their framework to nearly all languages and compilation phases of CompCert.

  Although the framework of Beringer *et al.* allows operational semantics of different languages to communicate, all of these languages should have a common memory model, namely the CompCert memory model. It would be interesting to investigate whether their framework can be generalized to support different memory models for the different languages involved. That way, the CH_2O memory model could be used on the level of the C sources and a low-level memory model on the level of the assembly.

- **Concurrency and atomics.** Shared-memory concurrency and atomic operations are the main omission from the C11 standard in the CH_2O semantics. Although shared-memory concurrency is a relatively new addition to the C and C++ standards, there is already a large body of ongoing work in this direction, see for example [SSO$^+$10, BOS$^+$11, SVN$^+$13, VBC$^+$15, BMN$^+$15]. These works have led to improvements of the standard text.

  There are still important open problems in the area of concurrent memory models, even for small sublanguages of C [BMN$^+$15]. Current memory models for these sublanguages involve just features specific to threads and atomic operations whereas we have focused on structs, unions, effective types and indeterminate memory. We hope that both directions are largely orthogonal and will eventually merge into a fully fledged C11 semantics.

  There is a strong correspondence between non-deterministic expression evaluation and concurrency as also indicated by our separation logic for expressions. It would be interesting to investigate non-deterministic evaluation in the context of a concurrent semantics for C, as being done by Memarian *et al.* in ongoing work [MNM+15]. Notice that concurrency in expressions is very weak, nearly any form of interference between subexpressions yields undefined behavior. This is different from shared-memory concurrency, where a semantics should provide stronger guarantees about interference between threads.

11.2.2 Correspondence with CompCert

The semantics of a programming language forms a contract between the programmer and the compiler. It would therefore be useful to investigate whether the results in this thesis could be applied to compiler verification, for example by formally relating the CompCert and CH_2O semantics. We discuss two possible directions.

First of all, it would be useful to establish a correspondence between the semantics of CH_2O core C and CompCert C. This correspondence should indicate that CompCert does not assign too many (undefined) behaviors and that CH_2O does not assign too few behaviors. For this approach it is important that the behaviors described by CompCert are indeed a subset of the behaviors described by CH_2O. In [KLW14] we have therefore described two extensions of CompCert based on CH_2O that succeed in giving a semantics to behaviors that were previously undefined in CompCert. As discussed in Section 6.4, it remains an open problem whether CompCert's treatment of struct and union values matches up with the CH_2O semantics.

A more challenging direction is to implement and verify a compiler frontend that translates CH_2O core C into an intermediate language of CompCert. This compiler frontend should perform more aggressive optimizations based on undefined behaviors related to effective types and sequence points.

11.2.3 Executable semantics

In this thesis we have presented an executable semantics written in Coq. We have used Coq's extraction mechanism to turn it into an interpreter that can compute all behaviors of a given program according to the C11 standard.

Although the majority of our interpreter has been implemented in Coq, it still uses some glue written in OCaml. Most notably, the translation of string literals and `printf` (that is currently just present for debugging purposes) is inaccurate. In order to implement these features properly, we need variadic functions and I/O.

We have tested our executable semantics on all examples in this thesis and a small test suite involving subtle corner cases of the C11 standard. The fact that we did not find any fundamental bugs in our semantics gives a strong indication that developing a semantics using a proof assistant is solid. Nonetheless, it would be interesting to test the CH_2O semantics against a more extensive test suite such as the GCC torture tests [GCC] or using a tool like CSmith [YCER11].

Our current executable semantics implements the operational semantics in a rather naive way. Most notably, the representation of the memory state is based on inefficient data structures (for example, array objects are represented as lists). In order to make the interpreter more robust, it would be useful to optimize the executable semantics by using more efficient data structures, or by contracting more redexes in one step.

Since we have implemented and proven correct the executable semantics in Coq, we can do so without sacrificing reliability.

11.2.4 Separation logic

We have presented a separation logic for C that has been proven sound with respect to the CH_2O operational semantics. Our separation logic correctly deals with features not considered by others such as expressions with side-effects, effective types, and the correct life-time of block scope local variables. In this section we list some directions for future work on our separation logic.

First of all, our separation logic currently abstracts away from object representations. It provides primitives that operate on the level of abstract values (mathematical integers, arrays, structs, *etc.*) rather than on the level of individual bytes or bits. This abstraction is desirable because it relieves one from having to deal with low-level details. However, in some situations one needs to reason about object representations explicitly. Since the granularity of our separation algebra for memories (Definition 8.1.6 on page 148) is already on the level of bits, we expect that one can extend our separation logic with primitives for object representations.

Secondly, in Section 8.7.3 we have presented a higher-order variant of our separation logic. We currently use higher-order separation logic solely to describe function specifications and to deal with recursion. It would be interesting to extend our separation logic with other higher-order features such as existential types to model data abstraction [BBT05] and recursive specifications as described in [DAH08, BRS^{+}11]. We do not foresee fundamental difficulties to add such higher-order features, since we expect these to be orthogonal to the features we have addressed in this thesis.

In order to make our separation logic practical for program verification, we need to implement a verification condition generator, which takes a program with logical annotations and generates a set of verification conditions that need to be verified. For traditional Hoare logic one often uses a variant of Dijkstra's weakest precondition calculus [Dij75] whereas for separation logic one often uses symbolic execution [BCO05]. We expect the extension of our separation logic as presented in Section 8.7 to provide opportunities for symbolic execution (possibly combined with a static analysis to determine the writable and read-only parts).

In traditional separation logic, there is a distinction between local variables and allocated storage [ORY01]. The value of each local variable is stored directly on the stack, whereas the memory is only used for allocated storage. As C supports pointers to local variables, our stack contains a reference to the actual value of each variable in memory. An advantage of distinguishing local variables from allocated storage is that assertions involving local variables can be written more compactly as the separating conjunction does not deal with those. It does not seem difficult to remove this level of indirection for the special case of local variables whose address is never taken. Also, since these variables cannot be accessed through pointers, one can statically decide if they could have been modified twice in the same expression and thereby if they could have caused undefined behavior due to a sequence point violation.

We also need automation to solve entailments involving assertions of separation logic in order to make our separation logic practically usable. Specific to formalizations of separation logic in Coq there has been work in this direction by for example Appel [App14], Chlipala [Chl11], and Bengtson *et al.* [BJB12].

11.3 Concluding remarks

Ritchie, who invented the C language with Thompson, once wrote [Rit93]:

> C is quirky, flawed, and an enormous success.

More than 20 years after Ritchie wrote the above, C and C++ remain among the most widely used programming languages in the world despite their flaws, and there is a good reason for that. C provides a good balance between low-level programming languages that allow for high portability and close control over the hardware, and high-level languages that allow for sophisticated compiler optimizations.

Although we believe that more modern functional programming languages are the preferred choice for many applications, we do not think they will replace C as a dominant general purpose programming language. It is thus important to pursue the use of proof assistants to get a better understanding of the C semantics and to apply proof assistants to verification of C programs.

Bibliography

[AAA$^+$11] Roberto Amadio, Andrea Asperti, Nicholas Ayache, Brian Campbell, Dominic P. Mulligan, Randy Pollack, Yann Régis-Gianas, Claudio Sacerdoti Coen, and Ian Stark. Certified Complexity. *Procedia CS*, 7:175–177, 2011.

[AAB$^+$13] Roberto M. Amadio, Nicholas Ayache, François Bobot, Jaap Boender, Brian Campbell, Ilias Garnier, Antoine Madet, James McKinna, Dominic P. Mulligan, Mauro Piccolo, Randy Pollack, Yann Régis-Gianas, Claudio Sacerdoti Coen, Ian Stark, and Paolo Tranquilli. Certified Complexity (CerCo). In *FOPARA*, pages 1–18, 2013.

[AB07] Andrew W. Appel and Sandrine Blazy. Separation Logic for Small-Step Cminor. In *TPHOLs*, volume 4732 of *LNCS*, pages 5–21, 2007.

[ABF$^+$05] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized Metatheory for the Masses: The PoplMark Challenge. In *TPHOLs*, volume 3603 of *LNCS*, pages 50–65, 2005.

[AM01] Andrew W. Appel and David A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *TOPLAS*, 23(5):657–683, 2001.

[AM13] Reynald Affeldt and Nicolas Marti. Towards formal verification of TLS network packet processing written in C. In *PLPV*, pages 35–46, 2013.

[AMRV07] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. A very modal model of a modern, major, general type system. In *POPL*, pages 109–122, 2007.

[ANS89] ANSI. *Programming language, C: ANSI X3.159-1989.* Number 160 in FIPS Publications. ANSI Technical Committee X3J11, 1989.

[App11] Andrew W. Appel. Verified Software Toolchain - (Invited Talk). In *ESOP*, volume 6602 of *LNCS*, pages 1–17, 2011.

[App14] Andrew W. Appel, editor. *Program Logics for Certified Compilers.* Cambridge University Press, 2014.

[App15] Andrew W. Appel. Verification of a Cryptographic Primitive: SHA-256, 2015.

[AS14] Reynald Affeldt and Kazuhiko Sakaguchi. An Intrinsic Encoding of a Subset of C and its Application to TLS Network Packet Processing. *JFR*, 7(1), 2014.

[BBC$^+$10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *CACM*, 53(2):66–75, 2010.

[BBT05] Bodil Biering, Lars Birkedal, and Noah Torp-Smith. BI Hyperdoctrines and Higher-Order Separation Logic. In *ESOP*, volume 3444 of *LNCS*, pages 233–247, 2005.

[BBW14] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. A Precise and Abstract Memory Model for C Using Symbolic Values. In *APLAS*, volume 8858 of *LNCS*, pages 449–468, 2014.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in TCS. Springer, 2004.

[BCC$^+$03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *PLDI*, pages 196–207, 2003.

[BCF$^+$14] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A trusted mechanised JavaScript specification. In *POPL*, pages 87–100, 2014.

[BCG$^+$08] Gilles Barthe, Pierre Crégut, Benjamin Grégoire, Thomas P. Jensen, and David Pichardie. The MOBIUS Proof Carrying Code Infrastructure. In *FMCO'07*, volume 5382 of *LNCS*, pages 1–24, 2008.

[BCO05] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic Execution with Separation Logic. In *APLAS*, volume 3780 of *LNCS*, pages 52–68, 2005.

[BCOP05] Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. Permission Accounting in Separation Logic. In *POPL*, pages 259–270, 2005.

[BDP12] Gilles Barthe, Delphine Demange, and David Pichardie. A Formally Verified SSA-Based Middle-End - Static Single Assignment Meets CompCert. In *ESOP*, volume 7211 of *LNCS*, pages 47–66, 2012.

[BHJM07] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast. *STTT*, 9(5-6):505–525, 2007.

[BHL$^+$10] Thomas Ball, Brian Hackett, Shuvendu K. Lahiri, Shaz Qadeer, and Julien Vanegue. Towards Scalable Modular Checking of User-Defined Properties. In *VSTTE*, volume 6217 of *LNCS*, pages 1–24, 2010.

[BJB12] Jesper Bengtson, Jonas Braband Jensen, and Lars Birkedal. Charge! - A Framework for Higher-Order Separation Logic in Coq. In *ITP*, volume 7406 of *LNCS*, pages 315–331, 2012.

[BJLM13] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. A Formally-Verified C Compiler Supporting Floating-Point Arithmetic. In *ARITH*, pages 107–115, 2013.

[BJSB11] Jesper Bengtson, Jonas Braband Jensen, Filip Sieczkowski, and Lars Birkedal. Verifying Object-Oriented Programs with Higher-Order Separation Logic in Coq. In *ITP*, volume 6898 of *LNCS*, pages 22–38, 2011.

[BLR11] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. A decade of software model checking with SLAM. *CACM*, 54(7):68–76, 2011.

[BM11] Sylvie Boldo and Guillaume Melquiond. Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq. In *ARITH*, pages 243–252, 2011.

[BMN$^+$15] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. The Problem of Programming Language Concurrency Semantics. In *ESOP*, volume 9032 of *LNCS*, pages 283–307, 2015.

[BMSW10] Sascha Böhme, Michał Moskal, Wolfram Schulte, and Burkhart Wolff. HOL-Boogie — An Interactive Prover-Backend for the Verifying C Compiler. *JAR*, 44(1–2):111–144, 2010.

[BOS$^+$11] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *POPL*, pages 55–66, 2011.

[Boy03] John Boyland. Checking Interference with Fractional Permissions. In *SAS*, volume 2694 of *LNCS*, pages 55–72, 2003.

[BRS$^+$11] Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. Step-indexed kripke models over recursive worlds. In *POPL*, pages 119–132, 2011.

[Bru72] N. G. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, pages 381–392, 1972.

[BSDA14] Lennart Beringer, Gordon Stewart, Robert Dockins, and Andrew W. Appel. Verified Compilation for Shared-Memory C. In *ESOP*, volume 8410 of *LNCS*, pages 107–127, 2014.

[BW96] Paul E. Black and Phillip J. Windley. Inference Rules for Programming Languages with Side Effects in Expressions. In *TPHOLs*, volume 1125 of *LNCS*, pages 51–60, 1996.

[Cam12] Brian Campbell. An Executable Semantics for CompCert C. In *CPP*, volume 7679 of *LNCS*, pages 60–75, 2012.

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, pages 238–252, 1977.

[CDH$^+$09] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A Practical System for Verifying Concurrent C. In *TPHOLs*, volume 5674 of *LNCS*, pages 23–42, 2009.

[CH88] Thierry Coquand and Gérard P. Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, 1988.

[Chl11] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*, pages 234–245. ACM, 2011.

[CHRS14] Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. End-to-end verification of stack-space bounds for C programs. In *PLDI*, page 30, 2014.

[CKK$^+$12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C - A Software Analysis Perspective. In *SEFM*, volume 7504 of *LNCS*, pages 233–247, 2012.

[CMTS09] Ernie Cohen, Michał Moskal, Stephan Tobies, and Wolfram Schulte. A Precise Yet Efficient Memory Model For C. *ENTCS*, 254:85–103, 2009.

[Coq15] Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2015. Available at `https://coq.inria.fr/doc/`.

[Cor09] Jonathan Corbet. Fun with NULL pointers, part 1, 2009. Blog post, available at `http://lwn.net/Articles/342330/`.

[COY07] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local Action and Abstract Separation Logic. In *LICS*, pages 366–378, 2007.

[CP90] Thierry Coquand and Christine Paulin. Inductively defined types. In *COLOG'88*, volume 417 of *LNCS*, pages 50–66, 1990.

[CS94] Jeffrey Cook and Sakthi Subramanian. A Formal Semantics for C in Nqthm. Technical Report 517D, Trusted Information Systems, 1994.

[DA13] Josiah Dodds and Andrew W. Appel. Mostly Sound Type System Improves a Foundational Program Verifier. In *CPP*, volume 8307 of *LNCS*, pages 17–32, 2013.

[DAH08] Robert Dockins, Andrew W. Appel, and Aquinas Hobor. Multimodal Separation Logic for Reasoning About Operational Semantics. *ENTCS*, 218:5–20, 2008.

[Dar09] Zaynah Dargaye. *Vérification formelle d'un compilateur optimisant pour langages fonctionnels*. PhD thesis, Université Paris 7 Diderot, 2009.

[Del00] David Delahaye. A Tactic Language for the System Coq. In *LPAR*, volume 1955 of *LNCS*, pages 85–95, 2000.

[DHA09] Robert Dockins, Aquinas Hobor, and Andrew W. Appel. A Fresh Look at Separation Algebras and Share Accounting. In *APLAS*, volume 5904 of *LNCS*, pages 161–177, 2009.

[Dij68] Edsger W. Dijkstra. Cooperating sequential processes. In *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

[Dij75] Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *CACM*, 18(8):453–457, 1975.

[DLRA12] Will Dietz, Peng Li, John Regehr, and Vikram S. Adve. Understanding integer overflow in C/C++. In *ICSE*, pages 760–770, 2012.

[Ell12] Chucky Ellison. *A Formal Semantics of C with Applications.* PhD thesis, University of Illinois, 2012.

[ER12a] Chucky Ellison and Grigore Roşu. An Executable Formal Semantics of C with Applications, 2012. Slides, available at `http://fsl.cs.uiuc.edu/pubs/ellison-rosu-2012-popl-slides.pdf`.

[ER12b] Chucky Ellison and Grigore Roşu. An executable formal semantics of C with applications. In *POPL*, pages 533–544, 2012.

[FFKD87] Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker, and Bruce F. Duba. A syntactic theory of sequential control. *TCS*, 52:205–237, 1987.

[Flo67] Robert Floyd. Assigning meaning to programs. In *Mathematical Aspects of Computer Science*, number 19 in Proceedings of Symposia in Applied Mathematics, pages 19–32, 1967.

[Fox03] Anthony C. J. Fox. Formal Specification and Verification of ARM6. In *TPHOLs*, volume 2758 of *LNCS*, pages 25–40, 2003.

[GCC] GCC. The GNU Compiler Collection. Website, available at `http://gcc.gnu.org/`.

[GH93] Yuri Gurevich and James K. Huggins. The Semantics of the C Programming Language. In *CSL*, volume 702 of *LNCS*, pages 274–308, 1993.

[GLAK14] David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. Don't Sweat the Small Stuff: Formal Verification of C Code Without the Pain. In *PLDI*, pages 429–439, 2014.

[GM10] Georges Gonthier and Assia Mahboubi. An introduction to small scale reflection in Coq. *JFR*, 3(2):95–152, 2010.

[Gur91] Yuri Gurevich. Evolving algebras: a tutorial introduction. *Bulletin of EATCS*, 43:264–284, 1991.

[GVF$^+$11] Liang Gu, Alexander Vaynberg, Bryan Ford, Zhong Shao, and David Costanzo. CertiKOS: a certified kernel for secure cloud computing. In *APSys*, page 3, 2011.

[HAN08] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle Semantics for Concurrent Separation Logic. In *ESOP*, volume 4960 of *LNCS*, pages 353–367, 2008.

[Her13] Paolo Herms. *Certification of a Tool Chain for Deductive Program Verification*. PhD thesis, Université Paris-Sud, 2013.

[HER15] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. Defining the Undefinedness of C. In *PLDI*, pages 336–345, 2015.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *CACM*, 12(10):576–580, 1969.

[Hob08] Aquinas Hobor. *Oracle Semantics*. PhD thesis, Princeton University, 2008.

[Hue97] Gérard P. Huet. The Zipper. *JFP*, 7(5):549–554, 1997.

[IEE08] IEEE Computer Society. *754-2008: IEEE Standard for Floating Point Arithmetic*. IEEE, 2008.

[IEE15] IEEE spectrum. The 2015 Top Ten Programming Languages, 2015. Website, available at `http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages`.

[ISO] ISO. WG14 Defect Report Summary. Website, available at `http://www.open-std.org/jtc1/sc22/wg14/www/docs/`.

[ISO99] ISO. *ISO/IEC 9899-1999: Programming languages – C*. ISO Working Group 14, 1999.

[ISO03] ISO. Rationale for International Standard – Programming Languages – C, 2003. Revision 5.10.

[ISO12] ISO. *ISO/IEC 9899-2011: Programming languages – C*. ISO Working Group 14, 2012.

[JLB$^+$15] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A Formally-Verified C Static Analyzer. In *POPL*, pages 247–259, 2015.

[JP04] Bart Jacobs and Erik Poll. Java Program Verification at Nijmegen: Developments and Perspective. In *ISSS'03*, volume 3233 of *LNCS*, pages 134–153, 2004.

[JP08] Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical Report CW-520, Katholieke Universiteit Leuven, 2008.

[JPL12] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. Validating LR(1) Parsers. In *ESOP*, volume 7211 of *LNCS*, pages 397–416, 2012.

[K$^+$09] Gerwin Klein et al. seL4: formal verification of an OS kernel. In *SOSP*, pages 207–220, 2009.

[KH15] Jeehoon Kang and Chung-Kil Hur. Verification of Separate Compilation for CompCert, 2015. Available at `http://sf.snu.ac.kr/compcertsep`.

[KHM$^+$15] Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. A Formal C Memory Model Supporting Integer-Pointer Casts. In *PLDI*, pages 326–335, 2015.

[KKB12] Gerwin Klein, Rafal Kolanski, and Andrew Boyton. Mechanised Separation Algebra. In *ITP*, volume 7406 of *LNCS*, pages 332–337, 2012.

[KLW14] Robbert Krebbers, Xavier Leroy, and Freek Wiedijk. Formal C semantics: CompCert and the C standard. In *ITP*, volume 8558 of *LNCS*, pages 543–548, 2014.

[KN06] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine, and compiler. *TOPLAS*, 28(4):619–695, 2006.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1st edition, 1978.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 2nd edition, 1988.

[Kre13] Robbert Krebbers. Aliasing restrictions of C11 formalized in Coq. In *CPP*, volume 8307 of *LNCS*, 2013.

[Kre14a] Robbert Krebbers. An Operational and Axiomatic Semantics for Non-determinism and Sequence Points in C. In *POPL*, pages 101–112, 2014.

[Kre14b] Robbert Krebbers. Separation algebras for C verification in Coq. In *VSTTE*, volume 8471 of *LNCS*, pages 150–166, 2014.

[Kre15] Robbert Krebbers. A Formal C Memory Model for Separation Logic, 2015. Accepted with revisions at JAR.

[Kri65] Saul A. Kripke. Semantical analysis of intuitionistic logic I. *Formal Systems and Recursive Functions*, pages 92–130, 1965.

[Kri11] Neelakantan R. Krishnaswami. *Verifying Higher-Order Imperative Programs with Higher-Order Separation Logic*. PhD thesis, Carnegie Mellon University, 2011.

[KW13] Robbert Krebbers and Freek Wiedijk. Separation Logic for Non-local Control Flow and Block Scope Variables. In *FoSSaCS*, volume 7794 of *LNCS*, pages 257–272, 2013.

[KW14] Robbert Krebbers and Freek Wiedijk. N1793: Stability of indeterminate values in C11, 2014. Available at `http://open-std.org/jtc1/sc22/wg14/www/docs/n1793.pdf`.

[KW15] Robbert Krebbers and Freek Wiedijk. A Typed C11 Semantics for Interactive Theorem Proving. In *CPP*, pages 15–27, 2015.

[KWAH12] Matt Kaufmann and Jr. Warren A. Hunt. Towards a formal model of the x86 ISA. Technical Report TR-12-07, University of Texas at Austin, 2012.

[LABS12] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. The CompCert Memory Model, Version 2. Research report RR-7987, INRIA, 2012. Revised version available as Chapter 32 of [App14].

[LB08] Xavier Leroy and Sandrine Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *JAR*, 41(1):1–31, 2008.

[LB11] Andreas Lochbihler and Lukas Bulwahn. Animating the Formalised Semantics of a Java-Like Language. In *ITP*, volume 6898 of *LNCS*, pages 216–232, 2011.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*, pages 42–54, 2006.

[Ler09a] Xavier Leroy. A formally verified compiler back-end. *JAR*, 43(4):363–446, 2009.

[Ler09b] Xavier Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009.

[Let04] Pierre Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq.* PhD thesis, Université Paris-Sud, 2004.

[LG09] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009.

[Loc12] Andreas Lochbihler. *A Machine-Checked, Type-Safe Model of Java Concurrency: Language, Virtual Machine, Memory Model, and Verified Compiler*. PhD thesis, Karlsruher Institut für Technologie, 2012.

[Mac01] Nick Maclaren. What is an Object in C Terms?, 2001. Mailing list message, available at `http://www.open-std.org/jtc1/sc22/wg14/9350`.

[Mal15] Gregory Malecha. Coq-ext-lib, a library of Coq definitions, theorems, and tactics, 2015. Available at `https://github.com/coq-ext-lib`.

[MHJM13] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. System V Application Binary Interface, AMD64 Architecture Processor Supplement, 2013.

[MIT] MITRE Corporation. CVE – Common Vulnerabilities and Exposures. Website, available at `https://cve.mitre.org`.

[MLH07] Oleg Mürk, Daniel Larsson, and Reiner Hähnle. KeY-C: A Tool for Verification of C Programs. In *CADE*, volume 4603 of *LNCS*, pages 385–390, 2007.

[MM11] Yannick Moy and Claude Marché. *The Jessie plugin for Deduction Verification in Frama-C, Tutorial and Reference Manual*, 2011.

[MNM$^+$15] Kayvan Memarian, Kyndylan Nienhuis, Justus Matthiesen, Peter Sewell, and James Lingard. Cerberus: towards an Executable Semantics for Sequential and Concurrent C11, 2015. Draft, retrieved at April 17, 2015 via private communication with the authors.

[Mog89] Eugenio Moggi. Computational Lambda-Calculus and Monads. In *LICS*, pages 14–23, 1989.

[MS13] Jean-François Monin and Xiaomu Shi. Handcrafted Inversions Made Operational on Operational Semantics. In *ITP*, volume 7998 of *LNCS*, pages 338–353, 2013.

[MS15] Kayvan Memarian and Peter Sewell. What is C in practice? (Cerberus survey), 2015. Website, available at `http://www.cl.cam.ac.uk/~pes20/cerberus/`.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David Macqueen. *The Definition of Standard ML (revised edition)*. MIT Press, 1997.

[Myr08] Magnus O. Myreen. *Formal verification of machine-code programs*. PhD thesis, University of Cambridge, 2008.

[Nak00] Hiroshi Nakano. A Modality for Recursion. In *LICS*, pages 255–266, 2000.

[NMRW02] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC*, volume 2304 of *LNCS*, pages 213–228, 2002.

[Nor98] Michael Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, 1998.

[Nor99] Michael Norrish. Deterministic Expressions in C. In *ESOP*, volume 1576 of *LNCS*, pages 147–161, 1999.

[Nor08] Michael Norrish. A formal semantics for C++. Technical report, NICTA, 2008.

[OBNS11] Scott Owens, Peter Böhm, Francesco Zappa Nardelli, and Peter Sewell. Lem: A Lightweight Tool for Heavyweight Semantics. In *ITP*, volume 6898 of *LNCS*, pages 363–369, 2011.

[O'H04] Peter W. O'Hearn. Resources, Concurrency and Local Reasoning. In *CONCUR*, volume 3170 of *LNCS*, pages 49–67, 2004.

[Ohe01] David von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001.

[ORY01] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local Reasoning about Programs that Alter Data Structures. In *CSL*, volume 2142 of *LNCS*, pages 1–19, 2001.

[Pap98] Nikolaos Papaspyrou. *A Formal Semantics for the C Programming Language*. PhD thesis, National Technical University of Athens, 1998.

[Pau89] Christine Paulin-Mohring. Extracting F_ω's Programs from Proofs in the Calculus of Constructions. In *POPL*, pages 89–104, 1989.

[Pol98] Robert Pollack. How to Believe a Machine-Checked Proof. In *Twenty Five Years of Constructive Type Theory*, 1998.

[Pym02] David J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*, volume 26 of *Applied Logic Series*. Springer, 2002.

[RCC+12] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *PLDI*, pages 335–346, 2012.

[RDRL11] Tahina Ramananandro, Gabriel Dos Reis, and Xavier Leroy. Formal verification of object layout for C++ multiple inheritance. In *POPL*, pages 67–80, 2011.

[RF95] Jonathan G. Rossie and Daniel P. Friedman. An Algebraic Semantics of Subobjects. In *OOPSLA*, pages 187–199, 1995.

[Rit93] Dennis Ritchie. The Development of the C Language. In *History of Programming Languages Conference*, pages 201–208, 1993.

[Sam14] Miro Samek. Are We Shooting Ourselves in the Foot with Stack Overflow?, 2014. Blog post, available at `http://embeddedgurus.com/state-space/2014/02/`.

[SBCA15] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. Compositional CompCert. In *POPL*, pages 275–287, 2015.

[SMK13] Thomas Sewell, Magnus O. Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *PLDI*, pages 471–482, 2013.

[SN08] Konrad Slind and Michael Norrish. A Brief Overview of HOL4. In *TPHOLs*, volume 5170 of *LNCS*, pages 28–32, 2008.

[SNO+07] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Tom Ridge, Susmit Sarkar, and Rok Strnisa. Ott: effective tool support for the working semanticist. In *ICFP*, pages 1–12, 2007.

[SO08] Matthieu Sozeau and Nicolas Oury. First-Class Type Classes. In *TPHOLs*, volume 5170 of *LNCS*, pages 278–293, 2008.

[Soz10] Matthieu Sozeau. A New Look at Generalized Rewriting in Type Theory. *JFR*, 2(1), 2010.

[SSO+10] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *CACM*, 53(7):89–97, 2010.

[SvdW11] Bas Spitters and Eelis van der Weegen. Type Classes for Mathematics in Type Theory. *Mathematical Structures in Computer Science*, 21(4):795–825, 2011.

[SVN+13] Jaroslav Sevcík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *JACM*, 60(3):22, 2013.

[Tay08] Ian Lance Taylor. Signed Overflow, 2008. Blog post, available at `http://www.airs.com/blog/archives/120`.

[TIO] TIOBE Software. Programming Community Index. Website, available at `http://www.tiobe.com/content/paperinfo/tpci/`.

[TKN07] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In *POPL*, pages 97–108, 2007.

[VBC+15] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *POPL*, pages 209–220, 2015.

[vON02] David von Oheimb and Tobias Nipkow. Hoare Logic for NanoJava: Auxiliary Variables, Side Effects, and Virtual Methods Revisited. In *FME*, volume 2391 of *LNCS*, pages 89–105, 2002.

[Wad90] Philip Wadler. Comprehending Monads. In *LISP and Functional Programming*, pages 61–78, 1990.

[WB89] Philip Wadler and Stephen Blott. How to Make ad-hoc Polymorphism Less ad-hoc. In *POPL*, pages 60–76, 1989.

[WCC+12] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Undefined Behavior: What Happened to My Code? In *APSys*, 2012.

[WN04] Martin Wildmoser and Tobias Nipkow. Certifying Machine Code Safety: Shallow Versus Deep Embedding. In *TPHOLs*, volume 3223 of *LNCS*, pages 305–320, 2004.

[WPN08] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The Isabelle Framework. In *TPHOLs*, volume 5170 of *LNCS*, pages 33–38, 2008.

[YCER11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *PLDI*, pages 283–294, 2011.

[ZNMZ12] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *POPL*, pages 427–440, 2012.

Appendix A

Syntax

A.1 Operators

$$\begin{aligned}
\circledcirc_c \in \textsf{compop} &::= \texttt{==} \mid \texttt{<=} \mid \texttt{<}\\
\circledcirc_b \in \textsf{bitop} &::= \texttt{\&} \mid \texttt{|} \mid \texttt{\^{}}\\
\circledcirc_a \in \textsf{arithop} &::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%}\\
\circledcirc_s \in \textsf{shiftop} &::= \texttt{<<} \mid \texttt{>>}\\
\circledcirc \in \textsf{binop} &::= \circledcirc_c \mid \circledcirc_b \mid \circledcirc_a \mid \circledcirc_s\\
\circledcirc_u \in \textsf{unop} &::= \texttt{-} \mid \texttt{\~{}} \mid \texttt{!}\\
\alpha \in \textsf{assign} &::= := \mid \circledcirc{:=} \mid {:=}\circledcirc
\end{aligned}$$

A.2 Types

$$\begin{aligned}
k \in K &:= \text{Set of integer ranks}\\
t \in \textsf{tag} &:= \text{Strings denoting struct/union names}\\
f \in \textsf{funname} &:= \text{Strings denoting of function names}\\
si \in \textsf{signedness} &::= \textsf{signed} \mid \textsf{unsigned}\\
\tau_\textsf{i} \in \textsf{inttype} &::= si\ k\\
\tau_\textsf{b}, \sigma_\textsf{b} \in \textsf{basetype} &::= \tau_\textsf{i} \mid \tau_\textsf{p}{*} \mid \textsf{void}\\
\tau_\textsf{p}, \sigma_\textsf{p} \in \textsf{ptrtype} &::= \tau \mid \textsf{any} \mid \vec{\tau} \to \tau\\
\tau, \sigma \in \textsf{type} &::= \tau_\textsf{b} \mid \tau[n] \mid \textsf{struct}\ t \mid \textsf{union}\ t\\
\Gamma \in \textsf{env} &:= (\textsf{tag} \to_\textsf{fin} \textsf{list type}) \times && \text{(types of struct/union fields)}\\
&\ (\textsf{funname} \to_\textsf{fin} (\textsf{list type} \times \textsf{type})) && \text{(types of functions)}
\end{aligned}$$

A.3 Permissions

$$\mathcal{L}(A) := \{\blacklozenge, \lozenge\} \times A \qquad \text{(lockable separation algebra)}$$

$$\begin{aligned}
\mathcal{C}(A) &:= \mathbb{Q} \times A && \text{(counting separation algebra)}\\
\mathcal{T}^{t}_{T}(A) &:= A \times T \text{ with } t \in T && \text{(tagged separation algebra)}\\
\gamma \in \mathsf{perm} &:= \mathcal{L}(\mathcal{C}(\mathbb{Q})) + \mathbb{Q} &&
\end{aligned}$$

A.4 Memory model

$$\begin{aligned}
o \in \mathsf{index} &:= \text{Set of memory indices}\\
\Delta \in \mathsf{memenv} &:= \mathsf{index} \rightarrow_{\mathsf{fin}} (\mathsf{type} \times \mathsf{bool})\\
r \in \mathsf{refseg} &::= \xhookrightarrow{\tau[n]} i \mid \xhookrightarrow{\mathsf{struct}\ t} i \mid \xhookrightarrow{\mathsf{union}\ t}_{q} i \text{ with } q \in \{\circ, \bullet\}\\
\vec{r} \in \mathsf{ref} &:= \mathsf{list}\ \mathsf{refseg}\\
a \in \mathsf{addr} &::= (o : \tau, \vec{r}, i)_{\sigma >_{*} \sigma_{\mathsf{p}}}\\
p \in \mathsf{ptr} &::= \mathsf{NULL}\ \sigma_{\mathsf{p}} \mid a \mid f^{\vec{\tau} \mapsto \tau}\\
b \in \mathsf{bit} &::= \lightning \mid 0 \mid 1 \mid (\mathsf{ptr}\ p)_{i}\\
\mathbf{b} \in \mathsf{pbit} &:= \mathcal{T}^{\lightning}_{\mathsf{bit}}(\mathsf{perm})\\
w \in \mathsf{mtree} &::= \mathsf{base}_{\tau_{\mathsf{b}}}\ \vec{\mathbf{b}} \mid \mathsf{array}_{\tau}\ \vec{w} \mid \mathsf{struct}_{t}\ \overrightarrow{w\vec{\mathbf{b}}} \mid \mathsf{union}_{t}\ (i, w, \vec{\mathbf{b}}) \mid \overline{\mathsf{union}}_{t}\ \vec{\mathbf{b}}\\
m \in \mathsf{mem} &:= \mathsf{index} \rightarrow_{\mathsf{fin}} (\mathsf{mtree} \times \mathsf{bool} + \mathsf{type})\\
v_{\mathsf{b}} \in \mathsf{baseval} &::= \mathsf{indet}\,\tau_{\mathsf{b}} \mid \mathsf{nothing} \mid \mathsf{int}_{\tau_{\mathsf{i}}}\ x \mid \mathsf{ptr}\,p \mid \mathsf{byte}\,\vec{b}\\
v \in \mathsf{val} &::= v_{\mathsf{b}} \mid \mathsf{array}_{\tau}\ \vec{v} \mid \mathsf{struct}_{t}\ \vec{v} \mid \mathsf{union}_{t}\ (i, v) \mid \overline{\mathsf{union}}_{t}\ \vec{v}\\
\Omega \in \mathsf{lockset} &:= \mathcal{P}_{\mathsf{fin}}(\mathsf{index} \times \mathbb{N})
\end{aligned}$$

A.5 CH$_2$O core C

$$\begin{aligned}
l \in \mathsf{labelname} &:= \text{Strings denoting labels}\\
\nu \in \mathsf{lrval} &:= \mathsf{ptr} + \mathsf{val}\\
e \in \mathsf{expr} &::= x_{i} \mid [\nu]_{\Omega} && \text{(variables and constants)}\\
&\mid *e \mid \&e && \text{(l-value and r-value conversion)}\\
&\mid e \mathbin{._{\mathbf{l}}} r \mid e \mathbin{._{\mathbf{r}}} r && \text{(indexing of arrays, structs and unions)}\\
&\mid e_1[\vec{r} := e_2] && \text{(altering arrays, structs and unions)}\\
&\mid e_1\ \alpha\ e_2 \mid \mathsf{load}\ e && \text{(assignments and loading from memory)}\\
&\mid e(\vec{e}) \mid \mathsf{abort}\ \tau && \text{(function calls and undefined behavior)}\\
&\mid \mathsf{alloc}_{\tau}\ e \mid \mathsf{free}\ e && \text{(allocation and deallocation)}\\
&\mid \circledcirc_{u}\ e \mid e_1 \circledcirc e_2 \mid (\tau)e && \text{(unary, binary and cast operators)}\\
&\mid (e_1, e_2) \mid e_1\ ?\ e_2 : e_3 && \text{(comma and sequenced conditional)}\\
s \in \mathsf{stmt} &::= e \mid \mathsf{return}\ e && \text{(expression and \texttt{return} statements)}\\
&\mid \mathsf{goto}\ l \mid l : && \text{(\texttt{goto} and label)}
\end{aligned}$$

$$
\begin{aligned}
&\mid \mathsf{throw}\ n \mid \mathsf{catch}\ s && \text{(}\texttt{throw}\text{ and }\texttt{catch}\text{)}\\
&\mid \mathsf{skip} \mid s_1\ ;\ s_2 && \text{(the skip statement and composition)}\\
&\mid \mathsf{local}_\tau\ s && \text{(block scope local variable declaration)}\\
&\mid \mathsf{loop}\ s && \text{(infinite loop)}\\
&\mid \mathsf{if}\ (e)\ s_1\ \mathsf{else}\ s_2 && \text{(conditional statement)}\\
\delta \in \mathsf{funenv} := {}& \mathsf{funname} \to_{\mathsf{fin}} \mathsf{stmt}
\end{aligned}
$$

A.6 CH$_2$O core C states

$$
\begin{aligned}
\mathcal{E}_{\mathsf{s}} \in \mathsf{ectx}_{\mathsf{s}} ::= {}& *\Box \mid \&\Box \mid \Box\ \mathbf{._l}\ r \mid \Box\ \mathbf{._r}\ r \mid \Box[\vec{r} := e_2] \mid e_1[\vec{r} := \Box]\\
&\mid \Box\ \alpha\ e_2 \mid e_1\ \alpha\ \Box \mid \mathsf{load}\ \Box \mid \Box(\vec{e}) \mid e(\vec{e}_1\,\Box\,\vec{e}_2) \mid \mathsf{alloc}_\tau\ \Box \mid \mathsf{free}\ \Box\\
&\mid\ \circledcirc_u\ \Box \mid \Box \circledcirc e_2 \mid e_1 \circledcirc \Box \mid (\tau)\Box \mid \Box\ ?\ e_2 : e_3 \mid (\Box, e_2)\\
\mathcal{E} \in \mathsf{ectx} := {}& \mathsf{list}\ \mathsf{ectx}_{\mathsf{s}}\\
\mathcal{S}_{\mathsf{s}} \in \mathsf{sctx}_{\mathsf{s}} ::= {}& \mathsf{catch}\ \Box \mid \Box\,;s_2 \mid s_1\,;\Box \mid \mathsf{loop}\ \Box \mid \mathsf{if}\ (e)\ \Box\ \mathsf{else}\ s_2 \mid \mathsf{if}\ (e)\ s_1\ \mathsf{else}\ \Box\\
\mathcal{S}_{\mathsf{e}} \in \mathsf{sctx}_{\mathsf{e}} ::= {}& \Box \mid \mathsf{return}\ \Box \mid \mathsf{if}\ (\Box)\ s_1\ \mathsf{else}\ s_2\\
\mathcal{P}_{\mathsf{s}} \in \mathsf{ctx}_{\mathsf{s}} ::= {}& \mathcal{S}_{\mathsf{s}} \mid \mathsf{local}_{o:\tau}\ \Box \mid (\mathcal{S}_{\mathsf{e}}, e) \mid \mathsf{resume}\ \mathcal{E} \mid \mathsf{params}\ f\ \overrightarrow{o\tau}\\
\mathcal{P} \in \mathsf{ctx} := {}& \mathsf{list}\ \mathsf{ctx}_{\mathsf{s}}\\
\rho \in \mathsf{stack} := {}& \mathsf{list}\ (\mathsf{index} \times \mathsf{type})\\
d \in \mathsf{direction} ::= {}& \searrow \mid \nearrow \mid \curvearrowright l \mid \uparrow n \mid \upuparrows v\\
\phi_U \in \mathsf{undef} ::= {}& \lightning\,\mathcal{E}\langle e\rangle \mid \lightning\,\mathcal{S}_{\mathsf{e}}\langle [v]_\Omega\rangle\\
\phi \in \mathsf{focus} ::= {}& (d,\ s) \mid e \mid \overline{\mathsf{call}}\ f\ \vec{v} \mid \overline{\mathsf{return}}\ f\ v \mid \overline{\mathsf{undef}}\ \phi_U\\
S \in \mathsf{state} ::= {}& \mathbf{S}(\mathcal{P},\ \phi,\ m)
\end{aligned}
$$

A.7 CH$_2$O abstract C

$$
\begin{aligned}
x \in \mathsf{string} := {}& \text{Set of strings}\\
k \in \mathsf{cintrank} ::= {}& \mathsf{char} \mid \mathsf{short} \mid \mathsf{int} \mid \mathsf{long} \mid \mathsf{long\ long} \mid \mathsf{ptr}\\
si \in \mathsf{signedness} ::= {}& \mathsf{signed} \mid \mathsf{unsigned}\\
\tau_{\mathsf{i}} \in \mathsf{cinttype} ::= {}& si^?\ k\\
\tau \in \mathsf{ctype} ::= {}& \mathsf{void} \mid \mathsf{def}\,x \mid \tau_{\mathsf{i}} \mid \tau* \mid \overrightarrow{\tau\ x^?} \to \tau \mid \tau[e]\\
&\mid \mathsf{struct}\ x \mid \mathsf{union}\ x \mid \mathsf{enum}\,x \mid \mathsf{typeof}\,e\\
e \in \mathsf{cexpr} ::= {}& x \mid \mathsf{const}_{\tau_{\mathsf{i}}}\ z \mid \mathsf{string}\ \vec{z} && \text{(variables, integer and string constants)}\\
&\mid \mathsf{sizeof}\ \tau \mid \mathsf{alignof}\ \tau \mid \mathsf{offsetof}\ \tau\ x\\
&\mid \tau\,\mathsf{min} \mid \tau\,\mathsf{max} \mid \tau\,\mathsf{bits} && \text{(implementation-defined constants)}\\
&\mid \&e \mid *e && \text{(address of and dereference operator)}\\
&\mid e\ .\ x && \text{(indexing of structs and unions)}\\
&\mid e_1\ \alpha\ e_2 && \text{(assignments)}
\end{aligned}
$$

$$\begin{array}{rll}
 & \mid e(\vec{e}) \mid & \text{(function calls)} \\
 & \mid \mathsf{alloc}_{\tau}\ e \mid \mathsf{free}\ e & \text{(allocation and deallocation)} \\
 & \mid \circledcirc_{u}\ e \mid e_1 \circledcirc e_2 \mid (\tau)I & \text{(unary, binary and cast operators)} \\
 & \mid e_1\ \texttt{\&\&}\ e_2 \mid e_1\ \texttt{||}\ e_2 \mid (e_1, e_2) \mid e_1\ ?\ e_2 : e_3 & \text{(sequenced operators)} \\
r \in \mathsf{crefseg} ::= & [e] \mid .x & \\
I \in \mathsf{cinit} ::= & e \mid \{\overrightarrow{\vec{r} := I}\} & \\
sto \in \mathsf{cstorage} ::= & \mathsf{static} \mid \mathsf{extern} \mid \mathsf{auto} & \\
s \in \mathsf{cstmt} ::= & e \mid \mathsf{return}\ e^{?} & \text{(expression and } \texttt{return} \text{ statements)} \\
 & \mid \mathsf{goto}\ x \mid x : s & \text{(}\texttt{goto}\text{ and label)} \\
 & \mid \mathsf{break} \mid \mathsf{continue} & \text{(}\texttt{break}\text{ and }\texttt{continue}\text{)} \\
 & \mid \{s\} & \text{(block scope)} \\
 & \mid \overrightarrow{sto}\ \tau\ x := I^{?}\ ;\ s & \text{(local variable declaration)} \\
 & \mid \mathsf{typedef}\ x := \tau\ ;\ s & \text{(local typedef declaration)} \\
 & \mid \mathsf{skip} \mid s_1\ ;\ s_2 & \text{(the skip statement and composition)} \\
 & \mid \mathsf{while}(e)\ s \mid \mathsf{do}\ s\ \mathsf{while}(e) & \text{(}\texttt{while}\text{ loops)} \\
 & \mid \mathsf{for}(e_1\ ;\ e_2\ ;\ e_3)\ s & \text{(}\texttt{for}\text{ loop)} \\
 & \mid \mathsf{if}\ (e)\ s_1\ \mathsf{else}\ s_2 & \text{(conditional statement)} \\
d \in \mathsf{decl} ::= & \mathsf{struct}\ \overrightarrow{\tau\, x} \mid \mathsf{union}\ \overrightarrow{\tau\, x} & \text{(struct and union declarations)} \\
 & \mid \mathsf{enum}\ \overrightarrow{x := e^{?}} : \tau_{\mathsf{i}} & \text{(enum declaration)} \\
 & \mid \mathsf{typedef}\ \tau & \text{(global typedef declaration)} \\
 & \mid \mathsf{global}\ I^{?} : \overrightarrow{sto}\ \tau & \text{(global variable declaration)} \\
 & \mid \mathsf{fun}\ s : \overrightarrow{sto}\ \tau & \text{(function declaration)} \\
\Theta \in \mathsf{decls} := & \mathsf{list}\ (\mathsf{string} \times \mathsf{decl}) &
\end{array}$$

Summary

This thesis describes a formal specification of the sequential fragment of the C programming language based on the official description of the C language, the C11 standard. Our formal specification of C, which is named $\mathrm{CH_2O}$, is used for the development of technology that enables verification of C programs in a standards compliant and compiler independent way.

The C programming language is addressed in this thesis because it is both among the most widely used and among the most bug-prone programming languages. C is widely used because of the performance of programs, the control over system resources such as memory, and the fact that C programs run on nearly any computer platform. On the other hand, C is bug-prone due to its weak type system and the absence of run-time checks. Our formal specification of C can be used as the basis for formal verification of C programs. This way, one can develop reliable C programs without sacrificing the benefits of C.

All parts of our formal specification of C are defined in the Coq proof assistant. Coq ensures that all definitions comprising the formal specification (the *semantics*) are mathematically well-formed and enjoy desirable properties. Most importantly, Coq allows us to define multiple versions of the C semantics and to prove that these versions correspond to each other. By developing multiple versions of the semantics, we have considered C from different perspectives and have thereby obtained a higher confidence in the correctness of our specification. We have, based on the C11 standard, defined the following kinds of semantics for C in Coq and proved that these correspond to each other.

- **Operational semantics.** An operational semantics describes the behavior of programs using individual computational steps. It is generally used to reason about program transformations and to prove metatheoretical properties.
- **Executable semantics.** An executable semantics is an algorithmic version of the operational semantics that allows one to compute the set of behaviors of a given program. It is generally used for debugging and testing purposes.
- **Axiomatic semantics.** An axiomatic semantics allows one to reason about programs in a structured fashion. It is generally used to reason about concrete C programs. Our axiomatic semantics is based on separation logic, a variant of Hoare logic which allows one to reason effectively about programs that use pointers. Pointers are commonplace in programs written in C.

Numerous metatheoretical properties of the formal semantics are proven in order to validate the formal definitions. The code extraction mechanism of Coq is used to turn the executable semantics into an interpreter, which is able to run the semantics on a test suite that comprises actual C programs.

Our formal version of C11 covers a large part of the C language including delicate features such as the C type system, arrays and pointers, structs and unions, typedefs and enums, implicit type conversions, object representations, expressions with side-effects in the presence of non-deterministic expression evaluation, local variables in the presence of non-local control (**`goto`**, **`break`**, **`continue`**, **`return`** and unstructured **`switch`**) and initializers.

Formalizing C is challenging not just because of the number of features, but also because C is oriented towards being efficiently implementable rather than being abstract in a mathematical sense. Not only does the behavior of each C program depend on implementation-defined properties such as sizes and endianness of integers, but in order to abstract even further from implementation specific choices, the C11 standard assigns a set of possible behaviors to each program instead of a unique behavior. In case of semantically illegal programs (those that have *undefined behavior* in C terminology) this set contains *all* possible behaviors (including letting the program crash). Surprisingly, correctly describing the set of programs that have undefined behavior in a formal manner is the hardest part of formalizing C.

Many difficulties in describing undefined behavior are due to the interaction between *low-level* and *high-level* data access in C. Low-level data access involves unstructured and untyped byte representations, and high-level data access involves abstract values such as structs and unions. Compilers often use a high-level view of data access to justify optimizations whereas many programmers expect data access to behave in a low-level way. Optimizations based on type-based alias analysis (*effective types* in C terminology) are examples thereof. The semantics in this thesis is therefore built on top of a novel typed memory model based on trees to correctly model the interaction between the low-level and high-level view of data access.

Our formal specification of C is faithful to the C11 standard, which means it describes all undefined behaviors of C11. As a consequence, when one proves something about a given program with respect to our semantics, it should behave that way with *any* ostensibly C11 compliant compiler such as GCC or Clang.

Samenvatting

Dit proefschrift beschrijft een formele specificatie van het sequentiële gedeelte van de programmeertaal C. Deze specificatie, genaamd CH_2O, is gebaseerd op de officiële beschrijving van de C taal, namelijk de C11 standaard. In dit proefschrift hebben we onze specificatie van C gebruikt voor de ontwikkeling van verificatietechnologieën om C programma's op een compileronafhankelijke manier te verifiëren.

Dit proefschrift beschouwt de programmeertaal C omdat C zowel een van de meest gebruikte als een van de meest onveilige programmeertalen is. Het wordt veel gebruikt vanwege de snelheid van de programma's, de controle over systeembronnen zoals het geheugen, en het feit dat C programma's op alle computerplatformen draaien. Aan de andere kant is C zeer onveilig vanwege het zwakke typesysteem en de afwezigheid van veiligheidscontroles tijdens het uitvoeren van programma's. Onze specificatie van C kan worden gebruikt als basis voor formele verificatie van C programma's. Op deze manier kan men betrouwbare C programma's ontwikkelen zonder in te hoeven leveren op de voordelen van C.

Onze formele specificatie van C is volledig beschreven middels het Coq bewijssysteem. Coq garandeert dat onze specificatie van C (de *semantiek*) wiskundig correct is en aan zekere eisen voldoet. In het bijzonder maakt Coq het mogelijk om meerdere semantieken voor C te definiëren en deze aan elkaar gelijk te bewijzen. Door meerdere semantieken te ontwikkelen hebben we C vanuit meerdere standpunten bekeken en een groter vertrouwen in de correctheid van onze specificatie verkregen. Met behulp van Coq hebben we de volgende semantieken voor C, alle gebaseerd op de C11 standaard, ontwikkeld en aan elkaar gelijk bewezen.

- **Operationele semantiek.** Een operationele semantiek beschrijft het gedrag van programma's in termen van losse berekeningsstappen. Het wordt veelal gebruikt om te redeneren over programmatransformaties en om metatheoretische eigenschappen van de taal vast te stellen.
- **Executeerbare semantiek.** Een executeerbare semantiek is een algoritmische versie van de operationele semantiek die het mogelijk maakt om alle gedragingen van een programma uit te rekenen. Het wordt veelal gebruikt voor het testen en 'debuggen' van de formele specificatie.
- **Axiomatische semantiek.** Een axiomatische semantiek maakt het mogelijk om op een structurele manier over programma's te redeneren. Het wordt veelal gebruikt om programma's correct te bewijzen. Onze axiomatische semantiek is gebaseerd op separatielogica, een variant van Hoare logica die het mogelijk

maakt te redeneren over programma's die gebruik maken van pointers. Pointers zijn gemeengoed in C programma's.

Om deze semantieken te valideren hebben we veel metatheoretische eigenschappen bewezen. Daarnaast hebben we de faciliteit voor code-extractie van Coq gebruikt om onze executeerbare semantiek te testen op een collectie van echte C programma's.

Onze specificatie van C11 omvat een groot gedeelte van de C taal. Dit gedeelte bevat lastige aspecten zoals het typesysteem, arrays en pointers, structs en unions, typedefs en enums, impliciete type-conversies, object-representaties, expressies met zij-effecten in combinatie met nondeterministische expressie-evaluatie, lokale variabelen in combinatie met ongestructureerde besturingsinstructies (**`goto`**, **`break`**, **`continue`**, **`return`** en ongestructureerde **`switch`**) en 'initialisers'.

Het ontwikkelen van een formele specificatie van C is niet slechts uitdagend vanwege de vele aspecten van C, maar ook omdat C ontstaan is met het oog op efficiëntie in plaats van wiskundige abstractie. Het gedrag van elk C programma is afhankelijk van zogenaamde *implementation-defined* eigenschappen zoals de afmetingen van integers. Om verder te abstraheren van implementatie-specifieke keuzes beschrijft de C11 standaard voor elk programma niet één uniek gedrag, maar een verzameling van mogelijke gedragingen. In het geval van semantisch verkeerde programma's (degene die *undefined behavior* hebben) zijn - in overeenstemming met de C11 standaard - alle gedragingen toegestaan, inclusief het vastlopen van het programma. Het is verrassend dat het correct en formeel beschrijven van undefined behavior het meest uitdagende onderdeel van een formele specificatie van C is

Veel moeilijkheden bij het beschrijven van undefined behavior in C komen voort uit het samenspel tussen het *low-level* en *high-level* gegevensperspectief in C. Het low-level gegevensperspectief omvat ongestructureerde en getypeerde byte-representaties, en het high-level gegevensperspectief omvat abstracte waarden zoals structs en unions. Compilers maken gebruik van het high-level gegevensperspectief om optimalisaties te rechtvaardigen terwijl veel programmeurs verwachten dat C programma's zich volledig op een low-level manier gedragen. Dit probleem speelt bijvoorbeeld bij optimalisaties die op type-gebaseerde alias-analyse gebaseerd zijn (*effective types* in C terminologie). De semantiek in dit proefschrift is gebouwd op een nieuw geheugenmodel gebaseerd op getypeerde bomen om het samenspel tussen het low-level en high-level gegevensperspectief correct formeel te beschrijven.

Onze formele specificatie van de C taal is trouw aan de C11 standaard en beschrijft dus alle undefined behaviors van C11. Dit heeft als gevolg dat als men iets bewijst over een programma met betrekking tot onze semantiek, het programma zich op de voorgeschreven manier zou moeten gedragen met *elke* officiële C11 compiler zoals GCC of Clang.

Curriculum Vitae

Education and employment

1987 Born on 22 September, Groesbeek, The Netherlands.

1999–2005 High School (VWO), Nijmeegse Scholengemeenschap Groenewoud, Nijmegen, The Netherlands.

2005–2010 BSc in Computer Science, Radboud University, Nijmegen, The Netherlands. Graduated with *cum laude* distinction.
Thesis title: Uitdrukkingskracht van XML schema's (Expressiveness of XML schemata), supervised by Prof. dr. Herman Geuvers.

2008–2010 MSc in Computer Science, Radboud University, Nijmegen, The Netherlands. Graduated with *cum laude* distinction.
Thesis title: Classical logic, control calculi and data types, supervised by Prof. dr. Herman Geuvers and Dr. James McKinna.

2010–2011 Junior researcher at the Formath EU project, Radboud University, Nijmegen, The Netherlands.

2011–2015 PhD student in Computer Science, Radboud University, Nijmegen, The Netherlands.
Thesis title: The C standard formalized in Coq, supervised by Prof. dr. Herman Geuvers and Dr. Freek Wiedijk.

2015– Post-doctoral researcher, Aarhus University, Aarhus, Denmark.

Notable academic visits during PhD

2012 Participated in the Oregon Programming Languages Summer School.

2013 Visited Dr. Xavier Leroy at Inria Paris Rocquencourt for 2 months.

2014 Participated in the Institut Henri Poincaré thematic trimester on semantics of proofs and certified mathematics for 3 weeks.

2014 Visited Prof. dr. Andrew Appel at Princeton University for 1 month.

Titles in the IPA Dissertation Series since 2009

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean.* Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques.* Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time.* Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation.* Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML.* Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange.* Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web.* Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory.* Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis.* Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks.* Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement.* Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers.* Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics.* Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäußer. *Model Checking Nondeterministic and Randomly Timed Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics.* Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking Timed Automata.* Faculty of Science, Mathematics and Computer Science, RU. 2010-06

A. Nugroho. *The Effects of UML Modeling on the Quality of Software.* Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. *Kleene Coalgebra.* Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications.* Faculty of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. *Formal Models for Component Connectors.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

M.M. Jaghoori. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services.* Faculty of Mathematics and Natural Sciences, UL. 2010-11

R. Bakhshi. *Gossiping Models: Formal Analysis of Epidemic Protocols.* Faculty of Sciences, Department of Computer Science, VUA. 2011-01

B.J. Arnoldus. *An Illumination of the Template Enigma: Software Code Generation with Templates.* Faculty of Mathematics and Computer Science, TU/e. 2011-02

E. Zambon. *Towards Optimal IT Availability Planning: Methods and Tools.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

L. Astefanoaei. *An Executable Theory of Multi-Agent Systems Refinement.* Faculty of Mathematics and Natural Sciences, UL. 2011-04

J. Proença. *Synchronous coordination of distributed components.* Faculty of Mathematics and Natural Sciences, UL. 2011-05

A. Moralı. *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

M. van der Bijl. *On changing models in Model-Based Testing.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

C. Krause. *Reconfigurable Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-08

M.E. Andrés. *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2011-09

M. Atif. *Formal Modeling and Verification of Distributed Failure Detectors.* Faculty of Mathematics and Computer Science, TU/e. 2011-10

P.J.A. van Tilburg. *From Computability to Executability – A process-theoretic view on automata theory.* Faculty of Mathematics and Computer Science, TU/e. 2011-11

Z. Protic. *Configuration management for models: Generic methods for model comparison and model co-evolution.* Faculty of Mathematics and Computer Science, TU/e. 2011-12

S. Georgievska. *Probability and Hiding in Concurrent Processes.* Faculty of Mathematics and Computer Science, TU/e. 2011-13

S. Malakuti. *Event Composition Model: Achieving Naturalness in Runtime Enforcement.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-14

M. Raffelsieper. *Cell Libraries and Verification.* Faculty of Mathematics and Computer Science, TU/e. 2011-15

C.P. Tsirogiannis. *Analysis of Flow and Visibility on Triangulated Terrains.* Faculty of Mathematics and Computer Science, TU/e. 2011-16

Y.-J. Moon. *Stochastic Models for Quality of Service of Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-17

R. Middelkoop. *Capturing and Exploiting Abstract Views of States in OO Verification.* Faculty of Mathematics and Computer Science, TU/e. 2011-18

M.F. van Amstel. *Assessing and Improving the Quality of Model Transformations.* Faculty of Mathematics and Computer Science, TU/e. 2011-19

A.N. Tamalet. *Towards Correct Programs in Practice.* Faculty of Science, Mathematics and Computer Science, RU. 2011-20

H.J.S. Basten. *Ambiguity Detection for Programming Language Grammars.* Faculty of Science, UvA. 2011-21

M. Izadi. *Model Checking of Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-22

L.C.L. Kats. *Building Blocks for Language Workbenches.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2011-23

S. Kemper. *Modelling and Analysis of Real-Time Coordination Patterns.* Faculty of Mathematics and Natural Sciences, UL. 2011-24

J. Wang. *Spiking Neural P Systems.* Faculty of Mathematics and Natural Sciences, UL. 2011-25

A. Khosravi. *Optimal Geometric Data Structures.* Faculty of Mathematics and Computer Science, TU/e. 2012-01

A. Middelkoop. *Inference of Program Properties with Attribute Grammars, Revisited.* Faculty of Science, UU. 2012-02

Z. Hemel. *Methods and Techniques for the Design and Implementation of Domain-Specific Languages.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-03

T. Dimkov. *Alignment of Organizational Security Policies: Theory and Practice.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-04

S. Sedghi. *Towards Provably Secure Efficiently Searchable Encryption.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-05

F. Heidarian Dehkordi. *Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference.* Faculty of Science, Mathematics and Computer Science, RU. 2012-06

K. Verbeek. *Algorithms for Cartographic Visualization.* Faculty of Mathematics and Computer Science, TU/e. 2012-07

D.E. Nadales Agut. *A Compositional Interchange Format for Hybrid Systems: Design and Implementation.* Faculty of Mechanical Engineering, TU/e. 2012-08

H. Rahmani. *Analysis of Protein-Protein Interaction Networks by Means of Annotated Graph Mining Algorithms.* Faculty of Mathematics and Natural Sciences, UL. 2012-09

S.D. Vermolen. *Software Language Evolution.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-10

L.J.P. Engelen. *From Napkin Sketches to Reliable Software.* Faculty of Mathematics and Computer Science, TU/e. 2012-11

F.P.M. Stappers. *Bridging Formal Models – An Engineering Perspective.* Faculty of Mathematics and Computer Science, TU/e. 2012-12

W. Heijstek. *Software Architecture Design in Global and Model-Centric Software Development.* Faculty of Mathematics and Natural Sciences, UL. 2012-13

C. Kop. *Higher Order Termination.* Faculty of Sciences, Department of Computer Science, VUA. 2012-14

A. Osaiweran. *Formal Development of Control Software in the Medical Systems Domain.* Faculty of Mathematics and Computer Science, TU/e. 2012-15

W. Kuijper. *Compositional Synthesis of Safety Controllers.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-16

H. Beohar. *Refinement of Communication and States in Models of Embedded Systems.* Faculty of Mathematics and Computer Science, TU/e. 2013-01

G. Igna. *Performance Analysis of Real-Time Task Systems using Timed Automata.* Faculty of Science, Mathematics and Computer Science, RU. 2013-02

E. Zambon. *Abstract Graph Transformation – Theory and Practice.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-03

B. Lijnse. *TOP to the Rescue – Task-Oriented Programming for Incident Response Applications.* Faculty of Science, Mathematics and Computer Science, RU. 2013-04

G.T. de Koning Gans. *Outsmarting Smart Cards.* Faculty of Science, Mathematics and Computer Science, RU. 2013-05

M.S. Greiler. *Test Suite Comprehension for Modular and Dynamic Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-06

L.E. Mamane. *Interactive mathematical documents: creation and presentation.* Faculty of Science, Mathematics and Computer Science, RU. 2013-07

M.M.H.P. van den Heuvel. *Composition and synchronization of real-time components upon one processor.* Faculty of Mathematics and Computer Science, TU/e. 2013-08

J. Businge. *Co-evolution of the Eclipse Framework and its Third-party Plug-ins.* Faculty of Mathematics and Computer Science, TU/e. 2013-09

S. van der Burg. *A Reference Architecture for Distributed Software Deployment.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-10

J.J.A. Keiren. *Advanced Reduction Techniques for Model Checking.* Faculty of Mathematics and Computer Science, TU/e. 2013-11

D.H.P. Gerrits. *Pushing and Pulling: Computing push plans for disk-shaped robots, and dynamic labelings for moving points.* Faculty of Mathematics and Computer Science, TU/e. 2013-12

M. Timmer. *Efficient Modelling, Generation and Analysis of Markov Automata.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-13

M.J.M. Roeloffzen. *Kinetic Data Structures in the Black-Box Model.* Faculty of Mathematics and Computer Science, TU/e. 2013-14

L. Lensink. *Applying Formal Methods in Software Development.* Faculty of Science, Mathematics and Computer Science, RU. 2013-15

C. Tankink. *Documentation and Formal Mathematics — Web Technology meets Proof Assistants.* Faculty of Science, Mathematics and Computer Science, RU. 2013-16

C. de Gouw. *Combining Monitoring with Run-time Assertion Checking.* Faculty of Mathematics and Natural Sciences, UL. 2013-17

J. van den Bos. *Gathering Evidence: Model-Driven Software Engineering in Automated Digital Forensics.* Faculty of Science, UvA. 2014-01

D. Hadziosmanovic. *The Process Matters: Cyber Security in Industrial Control Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-02

A.J.P. Jeckmans. *Cryptographically-Enhanced Privacy for Recommender Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-03

C.-P. Bezemer. *Performance Optimization of Multi-Tenant Software Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2014-04

T.M. Ngo. *Qualitative and Quantitative Information Flow Analysis for Multi-threaded Programs.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-05

A.W. Laarman. *Scalable Multi-Core Model Checking.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-06

J. Winter. *Coalgebraic Characterizations of Automata-Theoretic Classes.* Faculty of Science, Mathematics and Computer Science, RU. 2014-07

W. Meulemans. *Similarity Measures and Algorithms for Cartographic Schematization.* Faculty of Mathematics and Computer Science, TU/e. 2014-08

A.F.E. Belinfante. *JTorX: Exploring Model-Based Testing.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-09

A.P. van der Meer. *Domain Specific Languages and their Type Systems.* Faculty of Mathematics and Computer Science, TU/e. 2014-10

B.N. Vasilescu. *Social Aspects of Collaboration in Online Software Communities.* Faculty of Mathematics and Computer Science, TU/e. 2014-11

F.D. Aarts. *Tomte: Bridging the Gap between Active Learning and Real-World Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2014-12

N. Noroozi. *Improving Input-Output Conformance Testing Theories.* Faculty of Mathematics and Computer Science, TU/e. 2014-13

M. Helvensteijn. *Abstract Delta Modeling: Software Product Lines and Beyond.* Faculty of Mathematics and Natural Sciences, UL. 2014-14

P. Vullers. *Efficient Implementations of Attribute-based Credentials on Smart Cards.* Faculty of Science, Mathematics and Computer Science, RU. 2014-15

F.W. Takes. *Algorithms for Analyzing and Mining Real-World Graphs.* Faculty of Mathematics and Natural Sciences, UL. 2014-16

M.P. Schraagen. *Aspects of Record Linkage.* Faculty of Mathematics and Natural Sciences, UL. 2014-17

G. Alpár. *Attribute-Based Identity Management: Bridging the Cryptographic Design of ABCs with the Real World.* Faculty of Science, Mathematics and Computer Science, RU. 2015-01

A.J. van der Ploeg. *Efficient Abstractions for Visualization and Interaction.* Faculty of Science, UvA. 2015-02

R.J.M. Theunissen. *Supervisory Control in Health Care Systems.*

Faculty of Mechanical Engineering, TU/e. 2015-03

T.V. Bui. *A Software Architecture for Body Area Sensor Networks: Flexibility and Trustworthiness.* Faculty of Mathematics and Computer Science, TU/e. 2015-04

A. Guzzi. *Supporting Developers' Teamwork from within the IDE.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-05

T. Espinha. *Web Service Growing Pains: Understanding Services and Their Clients.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-06

S. Dietzel. *Resilient In-network Aggregation for Vehicular Networks.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-07

E. Costante. *Privacy throughout the Data Cycle.* Faculty of Mathematics and Computer Science, TU/e. 2015-08

S. Cranen. *Getting the point — Obtaining and understanding fixpoints in model checking.* Faculty of Mathematics and Computer Science, TU/e. 2015-09

R. Verdult. *The (in)security of proprietary cryptography.* Faculty of Science, Mathematics and Computer Science, RU. 2015-10

J.E.J. de Ruiter. *Lessons learned in the analysis of the EMV and TLS security protocols.* Faculty of Science, Mathematics and Computer Science, RU. 2015-11

Y. Dajsuren. *On the Design of an Architecture Framework and Quality Evaluation for Automotive Software Systems.* Faculty of Mathematics and Computer Science, TU/e. 2015-12

J. Bransen. *On the Incremental Evaluation of Higher-Order Attribute Grammars.* Faculty of Science, UU. 2015-13

S. Picek. *Applications of Evolutionary Computation to Cryptology.* Faculty of Science, Mathematics and Computer Science, RU. 2015-14

C. Chen. *Automated Fault Localization for Service-Oriented Software Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-15

S. te Brinke. *Developing Energy-Aware Software.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-16

R.W.J. Kersten. *Software Analysis Methods for Resource-Sensitive Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2015-17

J.C. Rot. *Enhanced coinduction.* Faculty of Mathematics and Natural Sciences, UL. 2015-18

M. Stolikj. *Building Blocks for the Internet of Things.* Faculty of Mathematics and Computer Science, TU/e. 2015-19

D. Gebler. *Robust SOS Specifications of Probabilistic Processes.* Faculty of Sciences, Department of Computer Science, VUA. 2015-20

M. Zaharieva-Stojanovski. *Closer to Reliable Software: Verifying functional behaviour of concurrent programs.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-21

R.J. Krebbers. *The C standard formalized in Coq.* Faculty of Science, Mathematics and Computer Science, RU. 2015-22