DimSum: A Decentralized Approach to Multi-language Semantics and Verification

MICHAEL SAMMLER, MPI-SWS, Germany
SIMON SPIES, MPI-SWS, Germany
YOUNGJU SONG, MPI-SWS, Germany
EMANUELE D'OSUALDO, MPI-SWS, Germany
ROBBERT KREBBERS, Radboud University Nijmegen, The Netherlands
DEEPAK GARG, MPI-SWS, Germany
DEREK DREYER, MPI-SWS, Germany

Prior work on multi-language program verification has achieved impressive results, including the compositional verification of complex compilers. But the existing approaches to this problem impose a variety of restrictions on the overall structure of multi-language programs (*e.g.*, fixing the source language, fixing the set of involved languages, fixing the memory model, or fixing the semantics of interoperation). In this paper, we explore the problem of how to avoid such global restrictions.

Concretely, we present **DimSum**: a new, *decentralized* approach to multi-language semantics and verification, which we have implemented in the Coq proof assistant. Decentralization means that we can define and reason about languages independently from each other (as independent *modules* communicating via events), but also combine and translate between them when necessary (via a library of combinators).

We apply DimSum to a high-level imperative language Rec (with an abstract memory model and function calls), a low-level assembly language **Asm** (with a concrete memory model, arbitrary jumps, and syscalls), and a mathematical specification language Spec. We evaluate DimSum on two case studies: an **Asm** library extending Rec with support for pointer comparison, and a coroutine library for Rec written in **Asm**. In both cases, we show how DimSum allows the **Asm** libraries to be abstracted to Rec-level specifications, despite the behavior of the **Asm** libraries not being syntactically expressible in Rec itself. We also verify an optimizing multi-pass compiler from Rec to **Asm**, showing that it is compatible with these **Asm** libraries.

CCS Concepts: • **Theory of computation** → **Logic and verification**; **Operational semantics**.

Additional Key Words and Phrases: multi-language semantics, verification, compilers, non-determinism, separation logic, Iris, Coq

ACM Reference Format:
Michael Sammler, Simon Spies, Youngju Song, Emanuele D'Osualdo, Robbert Krebbers, Deepak Garg, and Derek Dreyer. 2023. DimSum: A Decentralized Approach to Multi-language Semantics and Verification. *Proc. ACM Program. Lang.* 7, POPL, Article 27 (January 2023), 31 pages. https://doi.org/10.1145/3571220

Authors' addresses: Michael Sammler, MPI-SWS, Saarland Informatics Campus, Germany, msammler@mpi-sws.org; Simon Spies, MPI-SWS, Saarland Informatics Campus, Germany, spies@mpi-sws.org; Youngju Song, MPI-SWS, Saarland Informatics Campus, Germany, youngju@mpi-sws.org; Emanuele D'Osualdo, MPI-SWS, Saarland Informatics Campus, Germany, dosualdo@mpi-sws.org; Robbert Krebbers, Radboud University Nijmegen, The Netherlands, mail@robbertkrebbers.nl; Deepak Garg, MPI-SWS, Saarland Informatics Campus, Germany, dg@mpi-sws.org; Derek Dreyer, MPI-SWS, Saarland Informatics Campus, Germany, dreyer@mpi-sws.org.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2023 Copyright held by the owner/author(s).
2475-1421/2023/1-ART27
https://doi.org/10.1145/3571220

1 INTRODUCTION

To focus and simplify the problem of program verification, it is common to assume that the programs one is verifying are written in a single, well-defined language. However, many (if not most) real-world programs are assembled from components written in *multiple* languages. For example, programs in languages as diverse as Go, OCaml, Python, Rust, and Swift depend on standard or legacy libraries written in C; operating systems commonly implement interrupt handling in assembly code; low-level drivers link with architecture-specific assembly code. It thus remains a grand challenge to build formal methods that can handle such realistic multi-language programs.

What makes this so difficult is that, to verify a multi-language program, it is often not sufficient to verify the program's components separately—we have to additionally reason about the interactions between them. In particular, at the boundaries, we have to account for the friction that arises from the *language differences*. For example, in a high-level language calling other code is often done through a function call construct with argument names, whereas assembly-like languages typically use jumps and designated argument registers. The languages could also differ in their representation of values (*e.g.*, hierarchical vs. flat), their language features (*e.g.*, structured vs. unstructured control flow), and their memory models (*e.g.*, an abstract memory model where pointers are offsets into abstract blocks vs. a concrete memory model where pointers are concrete integers).

Much prior work on multi-language verification has focused on the specific important case of compiler verification, and in particular so-called *compositional compiler verification* [Neis et al. 2015; Perconti and Ahmed 2014; Stewart et al. 2015; Song et al. 2020; Koenig and Shao 2021]. The broad goal of compositional compiler verification is to specify and verify compilers in terms of how they transform individual *libraries* in a program, so that different libraries may be correctly linked together even if they are produced by different verified compilers for potentially different languages. (This is in contrast to the original CompCert [Leroy 2006], for example, which was verified only as a compiler for whole programs.) Building on the ideas of Matthews and Findler [2007], Ahmed and collaborators [Ahmed and Blume 2011; Perconti and Ahmed 2014; Mates et al. 2019; Patterson et al. 2017, 2022] have subsequently recognized compositional compiler verification as an instance of the much broader problem of *multi-language semantics*: what is the right way to even *define* the behavior and interoperation of multi-language programs so as to best support verified linking of code from different languages and compilers?

In this paper, we propose a new approach to multi-language semantics and verification, which we realize in a new Coq-based framework we call **DimSum**. Our approach is based on a simple observation: if we consider the aforementioned work in the context of *multi-language semantics*, then certain aspects of the semantics are *fixed up front*, thus restricting the flexibility with which components from different languages can be composed together. In contrast, DimSum is what we call *decentralized*: the semantics of a library can be specified (and the library verified) without regard to the other libraries in the program—without even needing to know in what languages or under what memory models the other libraries are written.

1.1 Principles of Decentralization

To give a clearer sense of the motivation behind DimSum, let us now articulate four key principles that we aim to satisfy and explain the ways in which prior approaches do or do not satisfy them.

Principle #1: No fixed source language. Among the first to explore compositional compiler verification were Hur and collaborators [Benton and Hur 2009, 2010; Hur and Dreyer 2011; Hur et al. 2012]; they developed a line of work that culminated in Pilsner [Neis et al. 2015], a compositionally verified compiler from an ML-like source language to a low-level assembly target language, which showcased the ability to soundly link the verified compilations of source-language modules with

tricky, handwritten assembly modules. Despite the sophistication of the Pilsner verification, a key limitation of the approach used by this line of work was identified by Perconti and Ahmed [2014]: Pilsner (and the other compilers in its lineage) only permit compiled libraries to be linked with assembly libraries for which there is *some semantically equivalent source module*. This limitation effectively rules out a significant use case for multi-language linking, since one of the main reasons to link compiled libraries against handwritten assembly libraries is when the latter provide some functionality that is *not* expressible in the source language of one's compiler.

Ahmed *et al.*'s line of work on multi-language semantics was at least partly motivated by the goal of lifting this restriction of Hur *et al.*'s work, a goal which Patterson and Ahmed [2019] later termed "source-independent linking". With DimSum, we aim to fulfill this goal as well: we do not fix any one language as "the source"; rather, we explicitly allow for the possibility of linking high-level code with low-level (*e.g.*, assembly) libraries that have no high-level semantic equivalent.

Principle #2: No fixed set of languages. Ahmed *et al.*'s aforementioned research programme on multi-language semantics takes the approach of combining all interoperating languages into one big "syntactic multi-language" and providing type-directed *wrappers* to convert values of one language to values of the other languages. One advantage of this approach is that it supports interoperation between libraries in very different languages—libraries which, unlike in Pilsner, are not expressible in any common source language. Another advantage is that compositional compiler correctness can then be formalized in terms of *contextual equivalence* (a very standard and well-understood criterion) in the syntactic multi-language.

A disadvantage of the syntactic multi-language approach, however, is that it requires one to fix the set of interoperating languages up front. As a result, it means that proofs about libraries in any one language must take into account all the other languages comprising the syntactic multi-language, and such proofs may break if new languages are added to the mix in the future.

With DimSum, we aim to support what we call *language-local* reasoning: we should not fix the set of interoperating languages up front, and we should be able to verify a library in one language without having to worry *a priori* about the other languages with which that library could be linked.

Principle #3: No fixed memory model. Since the development of CompCert, a wide range of projects have explored compositionally verified extensions of CompCert, including Compositional CompCert [Stewart et al. 2015], CompCertX [Gu et al. 2015; Wang et al. 2019], SepCompCert [Kang et al. 2016], CompCertM [Song et al. 2020], and CompCertO [Koenig and Shao 2021]. With the exception of SepCompCert, these projects follow Principles #1 and #2 above. However, unlike Pilsner and Ahmed *et al.*'s work, these CompCert extensions assume all interoperating languages to adhere to a particular memory model, namely the CompCert memory model.[1] As noted by Patterson and Ahmed [2019], this places a significant restriction on the set of languages that can realistically participate in a multi-language program.

With DimSum, we aim to support linking of libraries written in languages with *different* memory models, yet still allow such linking to be reasoned about in a language-local way (as per Principle #2). We will see a concrete instance of this problem in §2, where we link a language with an abstract memory model not unlike CompCert's (*i.e.*, pointers are abstract block ids with offsets) to an assembly language with a concrete memory model (*i.e.*, pointers are integer addresses).

Principle #4: No fixed notion of linking. A key aspect of multi-language semantics is formalizing inter-language *linking*. Individual languages typically come equipped with their own pre-existing notions of *syntactic* linking $L \cup L'$, and then on top of that, multi-language semantics

[1] Except for CompCertO, this assumption is crucial for the techniques. In the case of CompCertO, the assumption may not be crucial, but the approach has not been applied to a different memory model than the CompCert one. See §6 for details.

Program main	fn main() ≜ let x := yield(0) in print(x); let x := yield(0) in print(x); yield(0)
Library stream	fn stream(n) ≜ yield(n); stream($n+1$);
Library yield	yield : ... save and restore registers, and switch stack ...

Fig. 1. Example using coroutines.

frameworks often define their own notions of *semantic* linking $L \oplus L'$ in order to characterize interoperation between different languages. However, in all the work we are aware of, the definition of semantic linking is fixed up front.

In DimSum, we aim to avoid fixing any "official" notion of semantic linking up front; instead, we permit users of the framework to develop new, library-specific notions of linking that support higher-level reasoning principles. To illustrate what this looks like, let us consider a concrete example, depicted in Fig. 1: we take a high-level language with recursive functions called Rec and augment it with a coroutine library written in an assembly-like language called **Asm**.[2] More specifically, in the example, the two Rec-libraries stream and main are "linked" with each other through a coroutine **Asm**-library called **yield**. The stream function generates an infinite stream of integers 0, 1, 2, ... that is consumed by the main function (*i.e.*, the main function prints the first two elements and then returns the third). For the **Asm**-library **yield**, the exact implementation is not relevant. The only relevant aspect of **yield** is that it sequentially passes the control back-and-forth between main and stream whenever yield is called in either.[3]

Most approaches to multi-language semantics can reason about this program in one way or another. For example, what they could do is consider the **Asm**-program $\downarrow$main $\cup_{\mathsf{a}} \downarrow$stream $\cup_{\mathsf{a}}$ **yield** where $\downarrow \mathsf{R}$ denotes compilation and then show that it indeed prints 0, then 1, and then returns 2. What no existing approach can do—and here is where decentralization comes in—is locally extend the notion of semantic linking in one language (*e.g.*, Rec) due to the presence of a library written in another language (*e.g.*, the **yield**-library written in **Asm**). That is, at the level of Rec, all that we care about is that **yield** provides a *new form of semantic linking* "$\mathsf{R}_1 \oplus_{\mathsf{coro}} \mathsf{R}_2$", where function calls to yield on one side are perceived as function returns of yield on the other (*e.g.*, the call of yield(n) in stream is the return of yield(0) in main). This new, custom form of semantic linking "$\mathsf{R}_1 \oplus_{\mathsf{coro}} \mathsf{R}_2$" considerably simplifies reasoning about the interactions of R_1 and R_2, because we do not have to consider the **Asm**-implementation of **yield** itself. That is, whereas reasoning about **yield** drops down to the **Asm** level and involves reasoning about saving and restoring the stack pointer and certain other machine registers, reasoning about $\mathsf{R}_1 \oplus_{\mathsf{coro}} \mathsf{R}_2$ stays at the level of Rec.

1.2 DimSum

In this paper, we present DimSum, a Coq-based framework for multi-language semantics and verification that adheres to the four principles of decentralization laid out in §1.1. At the heart of DimSum lies our novel *decentralized multi-language semantics*, which forms the basis of all our reasoning. As a starting point for the semantics, we adopt the same viewpoint as the work surrounding CompCert [Stewart et al. 2015; Song et al. 2020; Koenig and Shao 2021], namely that the semantics of a library L written in language L is *a labeled transition system*, which we call *a module*. What makes the DimSum approach decentralized is how we reason about these modules.

[2] Inspired by Patrignani [2020], we depict Rec in red, sans-serif and **Asm** in **blue, bold**.
[3] We only consider a statically known set of coroutines so there is no function for spawning a coroutine. The first yield(0) in main starts the function stream with argument 0.

We take a page out of the work on process algebra and think of the modules as *communicating processes*. More specifically, we associate each language L with a set of events E_{L}, we give semantics to a library L as a module $[\![L]\!]_{\mathsf{L}} \in \mathsf{Module}(E_{\mathsf{L}})$, and then we model the interactions of modules (*e.g.*, jumps) as synchronization on events (*e.g.*, outgoing jumps are synchronized with incoming jumps). Following the style of process algebra, we build up larger modules from smaller ones using compositional combinators. For example, we define a suite of language-specific linking operators $M \oplus_{\mathsf{L}} M'$ that synchronize modules based on their events, and a collection of wrappers $\lceil M \rceil_{\mathsf{L} \rightleftharpoons \mathsf{L}'}$ that embed modules from one language L into another language L'.

The resulting approach to multi-language semantics is decentralized in the sense that when we reason about a particular collection of modules, we only care about their events and the languages to which these events belong. For example, in the rest of this paper, we consider modules in the high-level language Rec, the low-level assembly language **Asm**, and a mathematical specification language Spec. When we reason about the interactions of two Rec-modules $\mathsf{M}_1 \oplus_{\mathsf{r}} \mathsf{M}_2$ (*e.g.*, to prove that they refine a specification written in Spec), then we only need to know about the calling convention of Rec and its memory model. In particular, we do not need to take into account the existence of the language **Asm** or its memory model in any shape or form. In contrast, when we reason about an **Asm**-module $\mathbf{M_1}$ interacting with a Rec-module M_2, *i.e.*, $\mathbf{M_1} \oplus_{\mathsf{a}} \lceil \mathsf{M}_2 \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}$, we need to consider the different calling conventions and memory models of Rec and **Asm**. As a result, in DimSum, we can "mix and match" components written in different languages using a collection of language-specific combinators (*e.g.*, $\mathbf{M_1} \oplus_{\mathsf{a}} \mathbf{M_2}$, $\mathsf{M}_1 \oplus_{\mathsf{r}} \mathsf{M}_2$, $\mathsf{M}_1 \oplus_{\mathsf{coro}} \mathsf{M}_2$, and $\lceil \mathsf{M} \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}$).

To make the simple idea of "multi-language program components as communicating processes" scale to reasonably complex languages such as Rec and **Asm**, we bring several ideas from the literature to bear, albeit casting them in a new light:

(1) **Open-world events.** The work on fully abstract traces [Jeffrey and Rathke 2005; Laird 2007] introduced the idea of including all the visible parts of the program state in the events. Previously, the idea was used to prove contextual refinement via trace refinement where the traces consist of these detailed open-world events. In DimSum, we use similar open-world events to express the interactions of modules: modules share state (*e.g.*, the program memory), which has to be exchanged when two modules interact.

(2) **Wrappers.** The work on multi-language semantics [Matthews and Findler 2007; Ahmed and Blume 2011] introduced the idea of expressing language translations via wrappers. Previously, the idea was used to construct a syntactic multi-language using syntactic wrappers that embed expressions of other languages. In DimSum, we use wrappers at the level of the *semantics*: we define translations such as $\lceil \mathsf{M} \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}$ that operate on modules (*i.e.*, LTSs) and translate the events between two languages (*e.g.*, Rec and **Asm**).

(3) **Kripke relations.** The literature on compiler verification [Leroy and Blazy 2008; Hur and Dreyer 2011; Perconti and Ahmed 2014; Koenig and Shao 2021] employed the idea of *Kripke relations*—relations that maintain evolving internal state in order to track, *e.g.*, relationships between growing heaps—to reason about program executions. Previously, the idea was used to build expressive simulation relations for establishing compiler correctness. In DimSum, the role of expressive simulations is largely filled by our wrappers (see the previous idea) and, hence, we use Kripke relations to define them. Furthermore, to ease their formalization (in particular, to avoid explicit reasoning about possible worlds), we encode our Kripke relations in the separation logic Iris [Jung et al. 2015, 2018].

(4) **Rely-guarantee reasoning using angelic non-determinism.** The recent work on Conditional Contextual Refinement (CCR) [Song et al. 2023] explored the idea of expressing

rely-guarantee reasoning between a program component and its environment using angelic and demonic non-determinism. Previously, the idea was used to add user-provided preconditions and postconditions to contextual refinements. In DimSum, we apply the idea in our wrappers (*e.g.*, $\lceil M \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}$) to define language-specific protocols between a module and its environment.

Contributions. In summary, our main contribution is DimSum, a Coq-based framework for decentralized multi-language semantics and verification. The framework introduces the notion of a module, refinement between modules, and a library of language-agnostic combinators for linking and translating modules (§3). We then apply the framework to concrete instantiations:

- An instantiation of DimSum with (1) a high-level imperative language Rec with structured values, function calls, and an abstract memory model; (2) an assembly language **Asm** based on registers, unstructured jumps and a concrete memory model; and (3) a mathematical specification language Spec, together with linking operators (§4) and wrappers (§5).
- Two **Asm** libraries that extend Rec with new kinds of functionality: A library for pointer comparison (§2) and the coroutine library described earlier (§4.3).
- A compositional multi-pass compiler from Rec to **Asm** (§5).

DimSum is fully mechanized in the Coq proof assistant using Interaction Trees [Xia et al. 2020], Iris [Jung et al. 2015; Krebbers et al. 2017a; Jung et al. 2018], and the Iris Proof Mode [Krebbers et al. 2017b, 2018]. The Coq development can be found in Sammler et al. [2023b].

Scope of the paper. This paper presents a first step towards exploring a decentralized approach for multi-language verification. As such, the paper focuses on the setting of a C-like language Rec and an assembly language **Asm**. This is similar to the compositional variants of CompCert, except that the two languages differ in their memory model and program components can interact with unstructured jumps at the **Asm** level (and, of course, that Rec and **Asm** are significantly simpler than the realistic languages used by CompCert). It would be interesting to consider languages with other features like closures, garbage collection, types, or concurrency in future work.

Additionally, we focus our attention on safety properties and do not prove liveness properties (similar to Sprenger et al. [2020], who use process algebra ideas for the verification of distributed systems). This restriction simplified the development of DimSum's model (in particular, the notion of refinement). We believe it should be possible to extend DimSum to support liveness reasoning, but we leave this to future work.

2 KEY IDEAS

To illustrate the key ideas of DimSum, let us consider a motivating example. We want to verify the following program using libraries depicted in Fig. 2:

$$\begin{aligned} &\mathsf{fn}\ \mathsf{main}() \triangleq \mathsf{local}\ x[3]; x[0] \leftarrow 1; x[1] \leftarrow 2; && \backslash\backslash\ x \mapsto [1,2,0] \\ &\qquad\qquad \mathsf{memmove}(x+1, x+0, 2); && \backslash\backslash\ x \mapsto [1,1,2] \\ &\qquad\qquad \mathsf{print}(x[1]); \mathsf{print}(x[2]) \end{aligned}$$

The program first initializes the local array x, then moves the contents of x by one to the right using memmove, and finally prints the last two elements of x. It is primarily written in Rec, our high-level language with recursive functions. The parts that are not written in Rec, print and locle, are written in **Asm**, our low-level assembly language, because—as we will soon see—they cannot be implemented in Rec.

The function memmove is inspired by the corresponding function in the C standard library. It takes in a source pointer s, a destination pointer d, and the number of elements n that should be

Library memmove

fn memmove$(d, s, n) \triangleq$ if locle(d, s) then memcpy$(d, s, n, 1)$ else memcpy$(d+n-1, s+n-1, n, -1)$

fn memcpy$(d, s, n, o) \triangleq$ if $0 < n$ then $d \leftarrow !s$; memcpy$(d + o, s + o, n - 1, o)$

Library locle

locle : sle **x0**, **x0**, **x1**; ret

Library print

print : mov **x8**, **PRINT**; syscall; ret

Fig. 2. Libraries written in Rec and **Asm**.

copied over. It then checks whether the source lies to the left or to the right of the destination in memory using the **Asm**-library **locle**. Depending on the outcome, memmove then copies the memory either front-to-back or back-to-front to the destination. The function memmove varies the copy direction to ensure that it does not step on its own toes: even if the two pointers overlap, memmove will never overwrite data that was supposed to be copied later.

The functions print and locle are implemented in **Asm**. The function locle simply compares its two arguments with less-or-equal and returns the result. The arguments of locle are in the first two registers **x0** and **x1**, because the calling convention of **Asm** is that the arguments are in **x0**-**x8**. The return value is stored in **x0**. The function print leaves the argument **x0** unchanged, stores the flag for printing in **x8**, and then triggers a *syscall* (*i.e.*, a call to the operating system).

What makes this example interesting is that the functions print and locle have to be implemented in **Asm** because they cannot be implemented in the high-level language Rec. For print, the reason is that it makes a syscall, which—similar to C—is not something Rec can do. For locle, the reason is that it compares two pointers. Comparing pointers in **Asm** is easy because they are "just" integers. In contrast, Rec cannot compare pointers natively because it uses an abstract, block-based memory model (inspired by CompCert [Leroy and Blazy 2008]). That is, conceptually, memory in Rec is a collection of unordered blocks, and a pointer consists of a *block id* and an *offset into the block*. Since the blocks are unordered, comparing pointers from different blocks (as memmove does when called with pointers into different blocks) does not make sense from the perspective of Rec.

Verification goal. Our end goal for this example will be to show that the entire program refines a top-level specification, which says that the program prints 1 and then 2. Let us make this goal more precise. The program consists of several Rec and **Asm**-libraries. To obtain a whole program, we thus have to *compile* the Rec-libraries to **Asm**-libraries and then *link* all the **Asm**-libraries together. We end up with the following program:

$$\textbf{onetwo} \triangleq \downarrow \text{main} \cup_a \downarrow \text{memmove} \cup_a \textbf{locle} \cup_a \textbf{print}$$

Here, main denotes a singleton library containing the main function from above, $\mathbf{A_1} \cup_a \mathbf{A_2}$ denotes syntactic linking in **Asm**, and $\downarrow R$ denotes compilation from of a Rec-library R to an **Asm**-library. The compiler will be explained in §5, but its exact definition is not relevant for this example.

For the **Asm**-program **onetwo**, we then want to show that it refines a specification $\textbf{onetwo}_{\text{spec}}$:

$$\textbf{onetwo} \preceq_{a,s} \textbf{onetwo}_{\text{spec}}$$

In DimSum, refinement is defined as a notion of simulation, roughly stating that each step of **onetwo** can be matched by zero or more steps of $\textbf{onetwo}_{\text{spec}}$ producing the same externally visible behavior (for details see §3.1). The specification $\textbf{onetwo}_{\text{spec}}$ is written in our specification language Spec, which we will discuss later in this section. Roughly speaking, the specification simply says that the program prints 1 and then 2.

$$\textbf{Events} \ni \textbf{e} ::= \textbf{Jump!}(\textbf{r}, \textbf{m}) \mid \textbf{Jump?}(\textbf{r}, \textbf{m}) \mid \textbf{Syscall!}(\textbf{v}_1, \textbf{v}_2, \textbf{m}) \mid \textbf{SyscallRet?}(\textbf{v}, \textbf{m})$$
$$\textbf{Memory} \ni \textbf{m} \triangleq \mathbb{Z} \xrightarrow{\text{fin}} \textbf{Val} \cup \{\#\}$$
$$\textbf{Registers} \ni \textbf{r} \triangleq \textbf{RegisterName} \to \textbf{Val}$$
$$\textbf{Val} \ni \textbf{v} \triangleq \mathbb{Z}$$
$$\textbf{RegisterName} \ni \textbf{x} \triangleq \{\textbf{x0}, \ldots, \textbf{x30}, \textbf{sp}, \textbf{pc}\}$$

$$\text{Events} \ni \text{e} ::= \text{Call!}(\text{f}, \overline{\text{v}}, \text{m}) \mid \text{Call?}(\text{f}, \overline{\text{v}}, \text{m}) \mid \text{Return!}(\overline{\text{v}}, \text{m}) \mid \text{Return?}(\overline{\text{v}}, \text{m})$$
$$\text{Memory} \ni \text{m} \triangleq \text{Loc} \xrightarrow{\text{fin}} \text{Val}$$
$$\text{Loc} \ni \ell ::= \{\text{blockid} : \text{Id}, \text{offset} : \mathbb{Z}\}$$
$$\text{Val} \ni \text{v} ::= z : \mathbb{Z} \mid b : \mathbb{B} \mid \ell : \text{Loc}$$
$$\text{FnName} \ni \text{f} \triangleq \text{String}$$

Fig. 3. Asm and Rec events.

2.1 Event-Based Semantics

To explain how we define and prove $\textbf{onetwo} \preceq_{\text{s}} \textbf{onetwo}_{\text{spec}}$, we have to turn to the core building block of DimSum: *modules*. Modules are how DimSum assigns meaning to every program component (*e.g.*, memmove, **locle**, and $\textbf{onetwo}_{\text{spec}}$). The entire approach is centered around modules: we define interpretations of syntactic libraries into semantic modules, we define refinement as a simulation on modules, we define wrappers between modules of different languages, and we define semantic linking operators as combinators on modules. We will make the notion of a module precise in §3. For now, it suffices to know that a module $M \in \text{Module}(E)$ is a labeled transition system emitting events from a language-specific set of events E.

Event-based communication. The set of events E of each module $M \in \text{Module}(E)$ varies from language to language. For a given language, these events formalize how modules interact with their environment. That is, following the school of process algebra, we model the interaction of program components as event-based communication (*i.e.*, synchronization on events). For example, the events of Rec are function calls, and the events of **Asm** are jumps and syscalls. To scale this simple idea to stateful languages like Rec and **Asm**, we borrow an idea from the work on fully abstract traces [Jeffrey and Rathke 2005; Laird 2007]: the events carry *a detailed description of the program state*. As we will see, this enables expressing linking between modules as event synchronization.

The events of Rec and **Asm** are shown in Fig. 3. Rec-modules (*i.e.*, modules with Rec-events) can emit function calls with $\text{Call!}(\text{f}, \overline{\text{v}}, \text{m})$ and accept incoming function calls with $\text{Call?}(\text{f}, \overline{\text{v}}, \text{m})$. In general, we distinguish between *outgoing events* (!) and *incoming events* (?). In both cases, the events include the function name f, the arguments $\overline{\text{v}}$, *and the entire memory* m. Rec-modules can return from calls with $\text{Return!}(\text{v}, \text{m})$ and accept returns from functions they called with $\text{Return?}(\text{v}, \text{m})$. In **Asm**, modules communicate using jumps: $\textbf{Jump!}(\textbf{r}, \textbf{m})$ and $\textbf{Jump?}(\textbf{r}, \textbf{m})$. These events contain the registers **r** (including the target address of the jump $\textbf{r}(\textbf{pc})$) and the program memory **m**. Additionally, **Asm**-modules can initiate syscalls with $\textbf{Syscall!}(\textbf{v}_1, \textbf{v}_2, \textbf{m})$ where $\textbf{v}_1$ is the syscall identifier (*e.g.*, $\textbf{PRINT} = 8$), $\textbf{v}_2$ is the argument of the syscall, and **m** the memory when performing the syscall. They can then receive control again from the operating system through $\textbf{SyscallRet?}(\textbf{v}, \textbf{m})$ with return value **v** and resulting memory **m**. (We assume a syscall calling convention where all registers except the return register **x0** are restored.)

By comparing the events of the two languages, we can quite succinctly see their differences—the differences that we have to deal with in the proof of $\textbf{onetwo} \preceq_{\text{s}} \textbf{onetwo}_{\text{spec}}$. First of all, the two languages use a different function call structure. Calls in Rec are always bracketed with first a call event and then a return event, while **Asm**-modules only emit and accept jumps, without distinguishing calls from returns. The second important distinction is that **Asm** can do syscalls, whereas programs written in Rec cannot. This means that for any **Asm** program doing a syscall,

there is no corresponding Rec program. The third distinction is that Rec uses a structured model of values: they can be integers, locations, or Booleans. In **Asm**, values can only be integers, so pointers and Booleans are represented as integers. And finally, Rec and **Asm** use very different memory models. In Rec the memory model is block based, whereas in **Asm**, the memory is simply a map from addresses, *i.e.*, integers, to integers.[4]

With the events of Rec and **Asm** at hand, let us turn to the semantics of their syntactic libraries. For each language, we define a *module semantics* $[\![-]\!]_-$ that maps syntactic libraries (*i.e.*, R and **A**) into semantic modules (*i.e.*, $[\![\mathrm{R}]\!]_{\mathrm{r}} \in \mathrm{Module}(\mathrm{Events})$ and $[\![\mathbf{A}]\!]_{\mathrm{a}} \in \mathrm{Module}(\mathbf{Events})$) based on the operational semantics of the language. The exact definitions of the module semantics will not be relevant for the rest of this section, so we postpone them to §4.

High-level specifications. Before we can start the verification of $\mathbf{onetwo} \preceq_{\mathrm{a},\mathrm{s}} \mathbf{onetwo}_{\mathrm{spec}}$, we first have to *define the specification* $\mathbf{onetwo}_{\mathrm{spec}}$. For this, we use the specification language Spec. We will formally define Spec in §3.2. For now, it suffices to know that Spec is a language with *co-inductively defined* syntax and the following constructs:

$$\mathrm{Spec}(E) \ni p ::=_{\mathrm{coind}} \mathrm{any} \mid \mathrm{vis}(e); p \mid \mathrm{assume}(\phi); p \mid \exists x : T; p(x) \mid \cdots \qquad (e \in E, \phi \in \mathit{Prop})$$

The construct any means the implementation can do anything, *i.e.*, its behavior is not specified further. The construct $\mathrm{vis}(e); p$ means the implementation emits the visible event $e \in E$ and afterwards behaves like p. Spec is parametric over the set of events E that the programs emits. The construct $\mathrm{assume}(\phi); p$ means the implementation must behave like p *if* the proposition ϕ is true; otherwise, it may have any behavior. Finally, the construct $\exists x : T; p$ means the implementation must non-deterministically choose some $x : T$ and then behave like $p(x)$. As we will see in §2.2, the fact that programs are defined co-inductively in Spec allows us to express unbounded loops in the specifications.

With Spec at hand, we can turn to the specification of our example program **onetwo**. Since **onetwo** is an **Asm**-library, its specification is stated using **Asm**-events such as jumps and syscalls:

$$\begin{aligned}\mathbf{onetwo}_{\mathrm{spec}} \triangleq\ & \exists \mathbf{r}, \mathbf{m_0}; \mathrm{vis}(\mathbf{Jump?(r, m_0)}); \mathrm{assume}(\mathbf{r(pc)} = a_{\mathrm{main}} \wedge \mathbf{has\_stack(r(sp), m_0)});\\ & \exists \mathbf{m_1}; \mathrm{vis}(\mathbf{Syscall!(PRINT, 1, m_1)}); \exists \mathbf{m_2}; \mathrm{vis}(\mathbf{SyscallRet?}(*, \mathbf{m_2})); \mathrm{assume}(\mathbf{m_2} = \mathbf{m_1});\\ & \mathrm{vis}(\mathbf{Syscall!(PRINT, 2, *)}); \mathrm{vis}(\mathbf{SyscallRet?}(*, *)); \mathrm{any}\end{aligned}$$

First, the program can accept any jump to it from the environment with $\mathbf{Jump?(r, m_0)}$. The following assume encodes that, during the verification of an implementation against this specification, one need only consider the choices of $\mathbf{r}$ and $\mathbf{m_0}$ where the environment decides to jump to the *start of the main function* (*i.e.*, the program counter **pc** points to the first instruction in main after compilation), and where the stack pointer **sp** points to a valid stack in memory $\mathbf{m_0}$ (because compiled Rec-libraries assume a stack). In this case, the implementation will perform a sequence of events: it will print 1 by emitting a syscall and wait for the operating system to return;[5] then, assuming that the print syscall did not change the memory, it will print 2, and again wait for the operating system to return. After this point, the specification (for simplicity) uses any so as not to constrain the program's behavior further.

[4]Technically, a memory address can also be part of a "guard page" denoted by #. An access to a guard page immediately and safely terminates the program. Each stack is followed by a guard page that is used to detect stack overflow [Tanenbaum and Bos 2014, Section 11.5].

[5]The return value of the syscall is irrelevant so we omit it using $*$. The $*$ notation is interpreted via non-deterministic choice, *i.e.*, $\mathrm{vis}(\mathbf{SyscallRet?}(*, *)); p$ is defined as $\exists \mathbf{v}; \exists \mathbf{m}; \mathrm{vis}(\mathbf{SyscallRet?(v, m)}); p$.

$$
\begin{aligned}
[\![\mathbf{onetwo}]\!]_{\mathsf{a}} &= [\![\downarrow \mathsf{main} \cup_{\mathsf{a}} \downarrow \mathsf{memmove} \cup_{\mathsf{a}} \mathbf{locle} \cup_{\mathsf{a}} \mathbf{print}]\!]_{\mathsf{a}} && (1)\\
&\leq [\![\downarrow \mathsf{main}]\!]_{\mathsf{a}} \oplus_{\mathsf{a}} [\![\downarrow \mathsf{memmove}]\!]_{\mathsf{a}} \oplus_{\mathsf{a}} [\![\mathbf{locle}]\!]_{\mathsf{a}} \oplus_{\mathsf{a}} [\![\mathbf{print}]\!]_{\mathsf{a}} && (2)\\
&\leq \lceil [\![\mathsf{main}]\!]_{\mathsf{r}} \rceil_{\mathsf{r}\rightleftharpoons\mathsf{a}} \oplus_{\mathsf{a}} \lceil [\![\mathsf{memmove}]\!]_{\mathsf{r}} \rceil_{\mathsf{r}\rightleftharpoons\mathsf{a}} \oplus_{\mathsf{a}} [\![\mathbf{locle}]\!]_{\mathsf{a}} \oplus_{\mathsf{a}} [\![\mathbf{print}]\!]_{\mathsf{a}} && (3)\\
&\leq \lceil [\![\mathsf{main}]\!]_{\mathsf{r}} \rceil_{\mathsf{r}\rightleftharpoons\mathsf{a}} \oplus_{\mathsf{a}} \lceil [\![\mathsf{memmove}]\!]_{\mathsf{r}} \rceil_{\mathsf{r}\rightleftharpoons\mathsf{a}} \oplus_{\mathsf{a}} \lceil [\![\mathsf{locle}_{\mathsf{spec}}]\!]_{\mathsf{s}} \rceil_{\mathsf{r}\rightleftharpoons\mathsf{a}} \oplus_{\mathsf{a}} [\![\mathbf{print}_{\mathsf{spec}}]\!]_{\mathsf{s}} && (4)\\
&\leq \lceil [\![\mathsf{main}]\!]_{\mathsf{r}} \oplus_{\mathsf{r}} [\![\mathsf{memmove}]\!]_{\mathsf{r}} \oplus_{\mathsf{r}} [\![\mathsf{locle}_{\mathsf{spec}}]\!]_{\mathsf{s}} \rceil_{\mathsf{r}\rightleftharpoons\mathsf{a}} \oplus_{\mathsf{a}} [\![\mathbf{print}_{\mathsf{spec}}]\!]_{\mathsf{s}} && (5)\\
&\leq \lceil [\![\mathsf{main} \cup_{\mathsf{r}} \mathsf{memmove}]\!]_{\mathsf{r}} \oplus_{\mathsf{r}} [\![\mathsf{locle}_{\mathsf{spec}}]\!]_{\mathsf{s}} \rceil_{\mathsf{r}\rightleftharpoons\mathsf{a}} \oplus_{\mathsf{a}} [\![\mathbf{print}_{\mathsf{spec}}]\!]_{\mathsf{s}} && (6)\\
&\leq \lceil [\![\mathsf{main}_{\mathsf{spec}}]\!]_{\mathsf{s}} \rceil_{\mathsf{r}\rightleftharpoons\mathsf{a}} \oplus_{\mathsf{a}} [\![\mathbf{print}_{\mathsf{spec}}]\!]_{\mathsf{s}} && (7)\\
&\leq [\![\mathbf{onetwo}_{\mathsf{spec}}]\!]_{\mathsf{s}} && (8)
\end{aligned}
$$

Fig. 4. Proof outline.

Module refinement. Finally, we can see how our goal $\mathbf{onetwo} \preceq_{\mathsf{s}} \mathbf{onetwo}_{\mathsf{spec}}$ is defined:

$$\mathbf{onetwo} \preceq_{\mathsf{s}} \mathbf{onetwo}_{\mathsf{spec}} \triangleq [\![\mathbf{onetwo}]\!]_{\mathsf{a}} \preceq [\![\mathbf{onetwo}_{\mathsf{spec}}]\!]_{\mathsf{s}}$$

In DimSum, we do not center our reasoning around refinements relating the *syntax* of two programs (*i.e.*, libraries), but around refinements relating the *semantics* of two programs (*i.e.*, modules). Here, $[\![\cdot]\!]_{\mathsf{a}}$ is the module semantics of **Asm** mentioned earlier, $[\![\cdot]\!]_{\mathsf{s}}$ is the module semantics of Spec(**Events**) (defined in §3.2), and $M_1 \preceq M_2$ is *the language-agnostic simulation relation* of DimSum (defined in §3.1), which we use as the notion of refinement. Let us now see how to prove this refinement.

2.2 The Proof Strategy

The proof of $[\![\mathbf{onetwo}]\!]_{\mathsf{a}} \preceq [\![\mathbf{onetwo}_{\mathsf{spec}}]\!]_{\mathsf{s}}$ consists of a sequence of refinements, as depicted in Fig. 4. We can concatenate the sequence into our desired goal, because the refinement relation $M_1 \preceq M_2$ is transitive (and reflexive, see Lemma 3.1). The basic proof strategy—and here is where the decentralization of DimSum comes in—will be to decompose the program into several independent parts, gradually abstract those parts, and then assemble the entire program again. We will discuss these steps below. Along the way, we will point out whether the proof step is specific to the example or a generic reasoning principle for the involved languages (see Fig. 5).

Linking [(1) to (2), generic]. As a first step, we decompose the program **onetwo** into a collection of **Asm**-modules. We do so by replacing the *syntactic linking operator* $\mathbf{A_1} \cup_{\mathsf{a}} \mathbf{A_2}$ of **Asm** with the *semantic linking operator* $\mathbf{M_1} \,{}^{\mathbf{d_1}}\!\oplus_{\mathsf{a}}^{\mathbf{d_2}} \mathbf{M_2}$ of **Asm** (using ASM-LINK-SYN). The syntactic operator $\mathbf{A_1} \cup_{\mathsf{a}} \mathbf{A_2}$ takes two **Asm**-*libraries* $\mathbf{A_1}$ and $\mathbf{A_2}$ and combines their program code. In contrast, the semantic linking operator $\mathbf{M_1} \,{}^{\mathbf{d_1}}\!\oplus_{\mathsf{a}}^{\mathbf{d_2}} \mathbf{M_2}$ takes two **Asm**-*modules* $\mathbf{M_1}$ and $\mathbf{M_2}$ with associated instruction addresses $\mathbf{d_1}$ and $\mathbf{d_2}$ and then synchronizes them via their jump events. (We omit $\mathbf{d_1}$ and $\mathbf{d_2}$ where they clutter the discussion. We write $|\mathbf{A}|$ for the instruction addresses of $\mathbf{A}$.)

Let us take a closer look at the synchronization. Suppose we are linking two **Asm**-modules $\mathbf{M_1}$ and $\mathbf{M_2}$, and $\mathbf{M_1}$ is currently executing. If it wants to execute a jump, then it will emit the event $\mathbf{Jump!(r, m)}$ where the value of the program counter $\mathbf{r(pc)}$ indicates the destination. If the destination is in the instructions of $\mathbf{M_2}$, *i.e.*, in $\mathbf{d_2}$, then $\mathbf{M_2}$ gets to accept the jump event by emitting the dual event $\mathbf{Jump?(r, m)}$. In this case, the two components have synchronized, exchanging the values of the registers $\mathbf{r}$ and the memory $\mathbf{m}$. To the outside, the synchronization will be hidden: the combined module $\mathbf{M_1} \,{}^{\mathbf{d_1}}\!\oplus_{\mathsf{a}}^{\mathbf{d_2}} \mathbf{M_2}$ will do a silent τ-step. If the module $\mathbf{M_1}$ decides to jump *outside of* $\mathbf{M_2}$, *i.e.*, outside of $\mathbf{d_2}$, then $\mathbf{M_1} \,{}^{\mathbf{d_1}}\!\oplus_{\mathsf{a}}^{\mathbf{d_2}} \mathbf{M_2}$ will simply forward the jump to the environment.

$$\textsc{asm-link-syn}\quad \frac{|\mathbf{A}_1| \cap |\mathbf{A}_2| = \emptyset}{[\![\mathbf{A}_1 \cup_{\mathsf{a}} \mathbf{A}_2]\!]_{\mathsf{a}} \equiv [\![\mathbf{A}_1]\!]_{\mathsf{a}}\ {}^{|\mathbf{A}_1|}\oplus_{\mathsf{a}}^{|\mathbf{A}_2|}\ [\![\mathbf{A}_2]\!]_{\mathsf{a}}}$$

$$\textsc{rec-link-syn}\quad \frac{|\mathsf{R}_1| \cap |\mathsf{R}_2| = \emptyset}{[\![\mathsf{R}_1 \cup_{\mathsf{r}} \mathsf{R}_2]\!]_{\mathsf{r}} \equiv [\![\mathsf{R}_1]\!]_{\mathsf{r}}\ {}^{|\mathsf{R}_1|}\oplus_{\mathsf{r}}^{|\mathsf{R}_2|}\ [\![\mathsf{R}_2]\!]_{\mathsf{r}}}$$

$$\textsc{asm-link-horizontal}\quad \frac{\mathbf{M}_1 \preceq \mathbf{M}_1' \qquad \mathbf{M}_2 \preceq \mathbf{M}_2'}{\mathbf{M}_1\ {}^{\mathsf{d}_1}\oplus_{\mathsf{a}}^{\mathsf{d}_2}\ \mathbf{M}_2 \preceq \mathbf{M}_1'\ {}^{\mathsf{d}_1}\oplus_{\mathsf{a}}^{\mathsf{d}_2}\ \mathbf{M}_2'}$$

$$\textsc{rec-link-horizontal}\quad \frac{\mathsf{M}_1 \preceq \mathsf{M}_1' \qquad \mathsf{M}_2 \preceq \mathsf{M}_2'}{\mathsf{M}_1\ {}^{\mathsf{d}_1}\oplus_{\mathsf{r}}^{\mathsf{d}_2}\ \mathsf{M}_2 \preceq \mathsf{M}_1'\ {}^{\mathsf{d}_1}\oplus_{\mathsf{r}}^{\mathsf{d}_2}\ \mathsf{M}_2'}$$

$$\textsc{rec-wrapper-compat}\quad \frac{\mathsf{M} \preceq \mathsf{M}'}{\lceil \mathsf{M} \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}} \preceq \lceil \mathsf{M}' \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}}$$

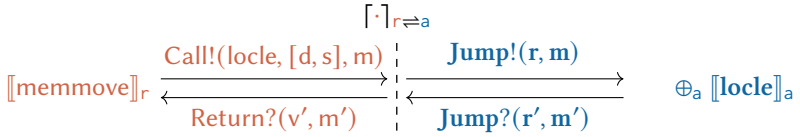
$$\textsc{compiler-correct}\quad \frac{\downarrow \mathsf{R}\ \textit{defined}}{[\![\downarrow \mathsf{R}]\!]_{\mathsf{a}} \preceq \lceil [\![\mathsf{R}]\!]_{\mathsf{r}} \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}}$$

$$\textsc{rec-to-asm-link}\quad \lceil \mathsf{M}_1 \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}} \oplus_{\mathsf{a}} \lceil \mathsf{M}_2 \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}} \preceq \lceil \mathsf{M}_1 \oplus_{\mathsf{r}} \mathsf{M}_2 \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}$$

Fig. 5. Proof rules of DimSum (where $M_1 \equiv M_2 \triangleq (M_1 \preceq M_2 \wedge M_2 \preceq M_1)$).

There is one additional, subtle property of the linking operator that is used in going from (1) to (2): *horizontal compositionality* of $\mathbf{M_1} \oplus_{\mathsf{a}} \mathbf{M_2}$ (ASM-LINK-HORIZONTAL). Horizontal compositionality in DimSum means compatibility with the refinement. We will see several semantic linking operators in DimSum (*i.e.*, $\mathbf{M_1} \oplus_{\mathsf{a}} \mathbf{M_2}$, $\mathsf{M}_1 \oplus_{\mathsf{r}} \mathsf{M}_2$, and $\mathsf{M}_1 \oplus_{\mathsf{coro}} \mathsf{M}_2$,) and they will all be horizontally compositional. In fact, all these linking operators will be derived from a single, language-generic linking operator that is horizontally compositional (see §3.3). But no need to get ahead of ourselves.

Module translation [(2) to (3), generic]. In the next step, we reap the benefits of the semantic linking operator: we can link modules that are not *syntactically* **Asm**-libraries, but *semantically* are **Asm**-modules. More precisely, in this step we take the **Asm**-modules $[\![\downarrow \mathsf{main}]\!]_{\mathsf{a}}$ and $[\![\downarrow \mathsf{memmove}]\!]_{\mathsf{a}}$ obtained through compilation of the Rec-libraries main and memmove, and then we turn them into Rec-modules $[\![\mathsf{main}]\!]_{\mathsf{r}}$ and $[\![\mathsf{memmove}]\!]_{\mathsf{r}}$ inside of a *semantic wrapper* $\lceil \cdot \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}$. The semantic wrapper $\lceil \cdot \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}$ is an embedding of Rec modules into **Asm** (*i.e.*, if M is an Rec module, then $\lceil \mathsf{M} \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}$ is an **Asm** module), and as such translates between Rec-events and **Asm**-events on the fly:



Conceptually, this wrapper is similar to a wrapper in a multi-language semantics of Matthews and Findler [2007]: it embeds constructs from one language into another. The key distinction of the wrappers in DimSum is that they are *semantic*: they operate on modules (*i.e.*, transition systems) instead of syntactic constructs. As a result, their task is to translate interactions (*i.e.*, events) between the two languages. Take the interaction of memmove and **locle** depicted above. In this case, the Rec module issues a call to the function locle with arguments d and s and the memory m. The wrapper $\lceil \cdot \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}$ then constructs the corresponding **Asm** jump event, including the correct representation of the registers **r** and of the memory **m**. When **locle** eventually jumps back, the wrapper translates the jump to a corresponding function return. (We will discuss these translations in more detail in §2.3.)

The wrapper $\lceil \cdot \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}$ has two important properties. The first (see COMPILER-CORRECT) is that the compiled program refines the source program wrapped by $\lceil \cdot \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}$—*i.e.*, our compiler is *correct* up to the translation of the wrapper. More specifically, a (syntactically) compiled Rec library $\downarrow \mathsf{R}$ behaves like the semantically translated source module $\lceil [\![\mathsf{R}]\!]_{\mathsf{r}} \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}$. The second (see REC-WRAPPER-COMPAT) is that the wrapper is compatible with refinement—this property will be used by the following steps.

$$
\begin{aligned}
\mathbf{print}_{\mathsf{spec}} \triangleq_{\mathsf{coind}}\ & \exists \mathbf{r}, \mathbf{m}; \mathsf{vis}(\mathbf{Jump?(r, m)}); \mathsf{assume}(\mathbf{r(pc)} = a_{\mathsf{print}}); \\
& \mathsf{vis}(\mathbf{Syscall!(PRINT, r(x0), m)}); \exists \mathbf{v}, \mathbf{m'}; \mathsf{vis}(\mathbf{SyscallRet?(v, m')}); \\
& \mathsf{vis}(\mathbf{Jump!(r[pc \mapsto r(x30)][x0 \mapsto v][x8 \mapsto *], m')}); \mathbf{print}_{\mathsf{spec}} \\
\mathsf{locle}_{\mathsf{spec}} \triangleq_{\mathsf{coind}}\ & \exists \mathsf{f}, \overline{\mathsf{v}}, \mathsf{m}; \mathsf{vis}(\mathsf{Call?(f, \overline{v}, m)}); \mathsf{assume}(\mathsf{f} = \mathsf{locle}); \mathsf{assume}(\overline{\mathsf{v}} \text{ is } [\ell_1, \ell_2]); \\
& \mathsf{if}\ \ell_1.\mathsf{blockid} = \ell_2.\mathsf{blockid}\ \mathsf{then}\ \mathsf{vis}(\mathsf{Return!}(\ell_1.\mathsf{offset} \le \ell_2.\mathsf{offset}, \mathsf{m})); \mathsf{locle}_{\mathsf{spec}} \\
& \mathsf{else}\ \exists \mathsf{b}; \mathsf{vis}(\mathsf{Return!(b, m)}); \mathsf{locle}_{\mathsf{spec}}
\end{aligned}
$$

Fig. 6. Specifications for **print** and **locle**.

Abstracting implementations [(3) to (4), example-specific]. In the next step, we replace the assembly libraries **print** and **locle** with high-level specifications written in Spec. We do so to abstract over the **Asm** implementation details of both libraries, since we only care about their *interaction behavior* with other modules (*e.g.*, their jumps, which values they compute, which syscalls they trigger). Formally, we prove:

$$\textsc{print-correct}\ [\![\mathbf{print}]\!]_{\mathsf{a}} \preceq [\![\mathbf{print}_{\mathsf{spec}}]\!]_{\mathsf{s}} \qquad \textsc{locle-correct}\ [\![\mathbf{locle}]\!]_{\mathsf{a}} \preceq \lceil [\![\mathsf{locle}_{\mathsf{spec}}]\!]_{\mathsf{s}} \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}$$

The specifications for **print** and **locle** are depicted in Fig. 6. Before even we consider the details of these specifications, we should discuss one fundamental difference that stands out: $\mathbf{print}_{\mathsf{spec}}$ is an **Asm**-level specification, while $\mathsf{locle}_{\mathsf{spec}}$ is a Rec-level specification. We can give a Rec specification to **locle**, because it has the *interaction behavior* of a Rec function. More precisely, while we cannot implement **locle** in Rec directly, we can still give it a Rec-level specification in Spec, because **locle** obeys the calling convention of Rec and triggers no syscalls. In contrast, the same cannot be said for **print**, because **print** does a syscall, which is beyond the interaction behavior of Rec.

Let us now turn to the details of both specifications. The specification $\mathbf{print}_{\mathsf{spec}}$ accepts jumps to the start of the print code address. Then it triggers a print syscall of the contents of **x0** and accepts the return value **v**, which is subsequently returned (by storing it in **x0**). The return address is then fetched from register **x30** and becomes the next program counter **pc**. Afterwards, the specification starts from the beginning again (*i.e.*, with $\mathbf{print}_{\mathsf{spec}}$). The last step is important to *reuse the module* for subsequent executions of **print**. It is made possible, because our programs in Spec are *co-inductive*, so $\mathbf{print}_{\mathsf{spec}}$ can refer to itself in its own definition.

The specification $\mathsf{locle}_{\mathsf{spec}}$ accepts an incoming function call to locle where the arguments are two locations ℓ_1 and ℓ_2. If the locations point to *the same block in memory* (*i.e.*, their block ids are the same), then $\mathsf{locle}_{\mathsf{spec}}$ compares their offsets and returns the result. Afterwards, the specification loops. If the locations point to *different blocks in memory*, then $\mathsf{locle}_{\mathsf{spec}}$ non-deterministically chooses a Boolean b and returns it. The non-deterministic choice here abstracts over the implementation detail of how exactly the Rec locations are mapped to the concrete **Asm** memory. This non-deterministic choice does not cause problems when verifying memmove since *s* and *d* cannot overlap if they point to different blocks and thus the result of locle is irrelevant.[6]

Leaving assembly behind [(4) to (6), generic]. In the next two steps, we exploit the fact that $[\![\mathsf{main}]\!]_{\mathsf{r}}$, $[\![\mathsf{memmove}]\!]_{\mathsf{r}}$, and $[\![\mathsf{locle}_{\mathsf{spec}}]\!]_{\mathsf{s}}$ obey the Rec interaction behavior: we lift them out of **Asm** to reason about them at the level of Rec in the next step. To do so, we introduce two Rec-linking operators: syntactic linking ($\mathsf{R}_1 \cup_{\mathsf{r}} \mathsf{R}_2$) and semantic linking ($\mathsf{M}_1 \,{}^{\mathsf{d}_1}\!\oplus^{\mathsf{d}_2}_{\mathsf{r}}\, \mathsf{M}_2$), analogous

[6] Our Coq development [Sammler et al. 2023b] additionally verifies a stronger version of $\mathsf{locle}_{\mathsf{spec}}$ that gives a consistent ordering of locations across multiple calls.

to **Asm**. (Here, d_1 and d_2 refer to the function names of M_1 and M_2 and we often omit them to avoid clutter.) We use the linking operators to combine the three Rec-modules into the module $[\![\mathsf{main} \cup_r \mathsf{memmove}]\!]_r \oplus_r [\![\mathsf{locle}_{\mathsf{spec}}]\!]_s$, leveraging that syntactic and semantic linking coincide for Rec-libraries (see REC-LINK-SYN), that $M_1 \oplus_r M_2$ is horizontally compositional (see REC-LINK-HORIZONTAL), and that the wrapper $\lceil\cdot\rceil_{r \rightleftharpoons a}$ is compatible with linking (see REC-TO-ASM-LINK).

In a typical verification task, we want to leave the level of assembly as much as possible. The reason is that it is simpler to reason about programs at the level of Rec than it is to reason about them at the level of **Asm**. In particular, when we reason about programs at the level of Rec, we do not have to think about the surrounding wrapper $\lceil\cdot\rceil_{r \rightleftharpoons a}$.

High-level reasoning [(6) to (7), example-specific]. In the next step, we can reap the benefits from reasoning at the level of Rec. More specifically, we can ignore that $[\![\mathsf{main}]\!]_r$, $[\![\mathsf{memmove}]\!]_r$, and $[\![\mathsf{locle}_{\mathsf{spec}}]\!]_s$ are inside of **Asm** (using the wrapper $\lceil\cdot\rceil_{r \rightleftharpoons a}$) and instead reason about *their interactions* at the level of Rec. We can abstract over their implementation details and show:

$$\textsc{main-correct}\ [\![\mathsf{main} \cup_r \mathsf{memmove}]\!]_r \oplus_r [\![\mathsf{locle}_{\mathsf{spec}}]\!]_s \preceq [\![\mathsf{main}_{\mathsf{spec}}]\!]_s$$

Here, $\mathsf{main}_{\mathsf{spec}}$ is a Spec specification for the three modules with Rec events:

$$\begin{aligned}\mathsf{main}_{\mathsf{spec}} \triangleq\ & \exists f, \overline{v}; \mathsf{vis}(\mathsf{Call?}(f, \overline{v}, *)); \mathsf{assume}(f = \mathsf{main}); \mathsf{assume}(\overline{v} = [\,]);\\ & \exists m_1; \mathsf{vis}(\mathsf{Call!}(\mathsf{print}, [1], m_1)); \exists m_2; \mathsf{vis}(\mathsf{Return?}(*, m_2)); \mathsf{assume}(m_2 = m_1);\\ & \exists m_3; \mathsf{vis}(\mathsf{Call!}(\mathsf{print}, [2], m_3)); \exists m_4; \mathsf{vis}(\mathsf{Return?}(*, m_4)); \mathsf{assume}(m_4 = m_3); \mathsf{any}\end{aligned}$$

The combined module will accept any incoming call to the main function. Subsequently, it will call print with argument 1 and some memory m_1, and expect print to return with the same memory. (Returning with a different memory $m_2 \neq m_1$ will be accepted, but in this case the specification does provide not any additional guarantees about the behavior of the program.) Subsequently, the specification will call print with argument 2, accept the corresponding return, and afterwards its behavior can be arbitrary. (Technically, this specification also needs to accept incoming calls after the call to print and behave arbitrarily in this case, but we omit this here for simplicity.)

Reasoning with specifications [(7) to (8), example-specific]. In a final step, we turn back to the $\mathbf{print}_{\mathsf{spec}}$ module. Recall that the module relies on interaction fundamentally not available at the level of Rec—syscalls—which is why we reason about it at the level of **Asm**. Fittingly, we also have to reason about $\lceil [\![\mathsf{main}_{\mathsf{spec}}]\!]_s \rceil_{r \rightleftharpoons a} \oplus_a [\![\mathbf{print}_{\mathsf{spec}}]\!]_s$ at the level of **Asm**. Typically, reasoning about programs at the level of **Asm** can be a daunting task, since there are many low-level details to consider. However, since we have already condensed the other modules into a single, specification $\mathsf{main}_{\mathsf{spec}}$, the last step is relatively straightforward:

$$\lceil [\![\mathsf{main}_{\mathsf{spec}}]\!]_s \rceil_{r \rightleftharpoons a} \oplus_a [\![\mathbf{print}_{\mathsf{spec}}]\!]_s \preceq [\![\mathbf{onetwo}_{\mathsf{spec}}]\!]_s$$

In the proof, we essentially only have to make sure that the calls in main and the jumps in **print** line up. The translation of the events is taken care of by the wrapper $\lceil\cdot\rceil_{r \rightleftharpoons a}$, which we discuss next.

2.3 Semantic Language Wrappers

One of the most important building blocks of the proof in §2.2 is the wrapper $\lceil\cdot\rceil_{r \rightleftharpoons a}$, which converts events from Rec to **Asm** *and back*. In this section, we take a closer look at how the wrapper works. Recall the event exchange between memmove and **locle** (in §2.2). In this exchange, the wrapper has to translate between (1) the calling conventions of both languages (*e.g.*, calls and returns in Rec are jumps in **Asm**), (2) the values of both languages (*e.g.*, structured values v in Rec are integers **v** in **Asm**), and (3) the memory models of both languages (*e.g.*, the block-based memory m in Rec is a flat memory **m** in **Asm**). As we will discuss below, the two key ingredients to getting this translation

$$\mathrm{Call!}(\mathsf{f}, \overline{\mathsf{v}}, \mathsf{m}) \rightharpoonup_w \mathbf{Jump!}(\mathbf{r}, \mathbf{m}) \triangleq \mathbf{r(pc)} = a_{\mathsf{f}} \land \overline{\mathsf{v}} \sim_w \mathbf{r(x0 \ldots x8)} \land |\overline{\mathsf{v}}| \leq 9 \land \mathsf{m} \sim_w \mathbf{m}$$

$$\mathrm{Return?}(\mathsf{v}, \mathsf{m}) \leftharpoonup_{\mathbf{r}', w} \mathbf{Jump?}(\mathbf{r}, \mathbf{m}) \triangleq \begin{array}{l} \mathbf{r(pc)} = \mathbf{r}'\mathbf{(x30)} \land \mathsf{v} \sim_w \mathbf{r(x0)} \land \mathsf{m} \sim_w \mathbf{m} \land \\ \mathbf{r(x19 \ldots x29, sp)} = \mathbf{r}'\mathbf{(x19 \ldots x29, sp)} \end{array}$$

Fig. 7. Select cases of the calling convention between Rec and **Asm**.

right are *Kripke relations* (explained using the direction Rec-to-**Asm**) and *angelic non-determinism* (explained using the direction **Asm**-to-Rec). In this section, we describe a simplified account of the wrapper $\lceil \cdot \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}$. Its actual definition is derived from the language generic combinators presented in §3.3, and can be found in the appendix [Sammler et al. 2023a, §E].

Kripke relations. Let us start with the direction of translating Rec events into **Asm** events (*e.g.*, translating $\mathrm{Call!}(\mathsf{locle}, [\mathsf{d}, \mathsf{s}], \mathsf{m})$ into $\mathbf{Jump!}(\mathbf{r}, \mathbf{m})$). In principle, this direction is relatively straightforward, because we go from a high-level language with more structure to a low-level language with less structure (*e.g.*, we map structured values v to integers $\mathbf{v}$). The main challenge in this direction is that the wrapper has to maintain a mapping from Rec-level block ids to **Asm**-level addresses, which remains consistent *across function calls*. That is, if we translate the location ℓ to the address $\mathbf{v}$ once, then we have to ensure that we pick $\mathbf{v}$ again for subsequent calls exposing ℓ, because assembly libraries typically expect the location ℓ to not move in between function calls.

To maintain a consistent mapping across function calls, the wrapper $\lceil \cdot \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}$ keeps around a block-id-to-address map w. In the full definition of the wrapper, this piece of state w is maintained using a separation logic relation (akin to the relations in §5). In the simplified account of $\lceil \cdot \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}$ that we discuss here, one can think of the map w as one component of the *internal state of the wrapper*. For example, in the case of translating outgoing calls to jumps, the wrapper transitions as follows:

$$\textsc{call-asm}\ \frac{\sigma \xrightarrow{\mathrm{Call!}(\mathsf{f},\overline{\mathsf{v}},\mathsf{m})}_M \sigma' \qquad \mathrm{Call!}(\mathsf{f}, \overline{\mathsf{v}}, \mathsf{m}) \rightharpoonup_{w'} \mathbf{Jump!}(\mathbf{r}, \mathbf{m}) \qquad w \subseteq w'}{(\mathsf{rec}, w, \sigma) \xrightarrow{\mathbf{Jump!}(\mathbf{r},\mathbf{m})}_{\lceil M \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}} (\mathbf{asm(r)}, w', \sigma')}$$

Here, the state of the wrapper contains information about who is currently executing (*e.g.*, rec or $\mathbf{asm(r)}$), the address mapping w, and the state of the wrapped module σ. (The reason why we record the registers $\mathbf{r}$ in $\mathbf{asm(r)}$ will become apparent below.) As the wrapper executes, the mapping w gradually grows along with the memories, written $w \subseteq w'$. In the context of Kripke relations, the state w is typically called a *world* and the relation $w \subseteq w'$ is *world extension*.

The relation $\mathrm{Call!}(\mathsf{f}, \overline{\mathsf{v}}, \mathsf{m}) \rightharpoonup_w \mathbf{Jump!}(\mathbf{r}, \mathbf{m})$ in 2.3, defined in Fig. 7, encodes a part of the calling convention of Rec and **Asm**. To define it, we first relate values between both languages:

$$z \sim_w z \qquad b \sim_w (\text{if } b \text{ then } 1 \text{ else } 0) \qquad \ell \sim_w w(\ell.\mathsf{blockid}) + \ell.\mathsf{offset}$$

In the case of locations ℓ, we look up the base address for the block in the mapping w. The relation can then be lifted to memories, written $\mathsf{m} \sim_w \mathbf{m}$. To translate a call from Rec to **Asm** with $\mathrm{Call!}(\mathsf{f}, \overline{\mathsf{v}}, \mathsf{m}) \rightharpoonup_w \mathbf{Jump!}(\mathbf{r}, \mathbf{m})$ we have to translate the components as follows: The program counter must point to the start address of the function f, the argument values must be stored in the registers $\mathbf{x0}$ to $\mathbf{x8}$, there may be at most nine arguments,[7] and the memories must be related. While this definition does incorporate quite a number of technical details about the calling conventions

[7]The calling convention of **Asm** restricts functions to nine registers. We rule out Rec functions with more than nine arguments in the compiler and restrict the number of function arguments in the wrapper.

of both languages, there is no way around it: when we call an **Asm** program, we have to make sure that its expectations are met, which includes satisfying the calling convention.

Angelic non-determinism. Let us now turn to the reverse direction (**Asm**-to-Rec). This direction is more challenging because we need to "guess" the additional structure of the representation at the level of Rec. For example, consider translating **Jump?(r, m)** to Return?(v, m) (*e.g.*, when **locle** returns from **Asm**). In this translation, the return value is stored in **x0** as *an integer* and we need to pick "the right" Rec return value v. The issue is that there can be multiple candidates, but not all will work. For instance, if **locle** returns **0** (*i.e.*, the first location is not less-or-equal to the second) and this **0** is subsequently translated to a location ℓ_0 instead of the Boolean false, then memmove will have undefined behavior. Unfortunately, the wrapper cannot choose the right v by itself, because locally, it knows possible candidates (*e.g.*, false, 0, and ℓ_0), but it does not know which one will work "down the road". (Also, Rec is untyped so there is no type system to help with this choice.) To help the wrapper out, we delegate the choice to a well-meaning angel: we use *angelic non-determinism* [Floyd 1967].

Before we discuss angelic non-determinism, let us first explain the calling convention for this direction (see Fig. 7). The relation for this case, $(\leftarrowtail_{\mathbf{r}',w})$, takes an additional piece of state: the register state $\mathbf{r}'$ that we record in $\mathbf{asm}(\mathbf{r}')$ when calling **Asm**-code (see 2.3). The relation requires the program counter to point to the original return address (in **x30**), the return values and memories to be related, and the callee-saved registers **x19**, ..., **x29**, **sp** to be restored.

Let us turn to *angelic non-determinism* and how it helps us here. To return from **Asm**, we have to define the analogue of 2.3 but for Return?(v, m) $\leftarrowtail_{\mathbf{r}',w}$ **Jump?(r, m)**. However, if we follow the structure of 2.3, then event translation would become a *proof obligation for the wrapper* including choosing v and m. That is, applying the hypothetical rule would lead to the obligation:

$$\text{“}\exists \mathsf{v}, \mathsf{m}.\ (\mathsf{Return?(v, m)} \leftarrowtail_{\mathbf{r}',w} \mathbf{Jump?(r, m)}) \wedge \ldots\text{”}$$

However, what we want here is that the event translation becomes *an assumption of the wrapper* including "the right choices" for v and m. In other words, we want something like:

$$\text{“}\forall \mathsf{v}, \mathsf{m}.\ (\mathsf{Return?(v, m)} \leftarrowtail_{\mathbf{r}',w} \mathbf{Jump?(r, m)}) \Rightarrow \ldots\text{”}$$

Unfortunately, we cannot literally define an analogue of 2.3 using this precondition, because then there could only be a single successor state σ' for all possible memories m and values v. There are, however, typically multiple candidates σ' depending on the choices of m and v. Since the wrapper does not know how to choose m and v itself, the only sensible option is to continue in *all possible states* σ' under the assumption of Return?(v, m) $\leftarrowtail_{\mathbf{r}',w}$ **Jump?(r, m)**. That is exactly what *angelic non-determinism* allows us to do. (We make formal how in the next section, §3.1.) It will then be the job of the angel to pick the right m and v, and thereby choose one of the states σ'.

Of course, we cannot keep delegating the responsibility to make "the right choices" to the angel forever. Eventually, we—as the user of DimSum—have to slip into the role of the angel and provide "the right choices". In this case, we do so in proving REC-TO-ASM-LINK (*i.e.*, $\lceil \mathsf{M}_1 \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}} \oplus_{\mathsf{a}} \lceil \mathsf{M}_2 \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}} \preceq \lceil \mathsf{M}_1 \oplus_{\mathsf{r}} \mathsf{M}_2 \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}$). Consider the case where M_2 returns to M_1. In terms of events, this means we come from Rec, go through **Asm**, and then return to Rec again. This path allows us as the user to observe the right choice: the memory m and value v will be determined by M_2 and we, as the angel, can then forward them to M_1. Inspired by CCR [Song et al. 2023], this use of angelic non-determinism allows us to express rely-guarantee protocols between modules and their environment.

3 MODULES AND REFINEMENT

In this section, we discuss the formal definition of modules and simulation (in §3.1), the meaning of non-deterministic choices (in §3.2), and the library of compositional combinators (in §3.3).

$$\frac{\sigma \in \Sigma}{\sigma \xrightarrow{\mathsf{nil}}{}^*_M \Sigma} \qquad \frac{\exists \Sigma'.\ \sigma \xrightarrow{\alpha_1}_M \Sigma' \wedge \forall \sigma' \in \Sigma'.\ \sigma' \xrightarrow{\overline{e_2}}{}^*_M \Sigma}{\sigma \xrightarrow{\alpha_1 ::_? \overline{e_2}}{}^*_M \Sigma}$$

Fig. 8. Multistep execution $\sigma \xrightarrow{\overline{e}}{}^* \Sigma$ with $\alpha_1 ::_? \overline{e_2} \triangleq \text{if } \alpha_1 = e_1 \text{ then } e_1 :: \overline{e_2} \text{ else } \overline{e_2}$.

3.1 Modules and Refinement in the Abstract

A module $M \in \mathsf{Module}(E)$ is a labeled transition system with events drawn from the set E and demonic and angelic non-determinism. Formally, a module $M = (S, \rightarrow, \sigma^0)$ consists of a set of states S, an initial state σ^0, and a transition relation $\rightarrow \in \mathcal{P}(S \times (E \uplus \{\tau\}) \times \mathcal{P}(S))$. The labels type $\alpha \in E \uplus \{\tau\}$ indicates that each transition can either emit a visible event $e \in E$ or be silent, denoted by τ. Notably, the transitions of a module $\sigma \xrightarrow{\alpha} \Sigma$ go from a single state σ to a *set of states* Σ. The use of a set is inspired by alternating automata [Chandra et al. 1981; Vardi 1995] and binary multi-relations [Rewitzky 2003] as a means to incorporate both demonic and angelic non-determinism. Demonic non-determinism works "as usual": a single state σ can transition to multiple sets Σ (*i.e.*, $\sigma \xrightarrow{\alpha} \Sigma$ and $\sigma \xrightarrow{\alpha'} \Sigma'$ where $\Sigma \neq \Sigma'$). Angelic non-determinism works differently: after the transition $\sigma \xrightarrow{\alpha} \Sigma$, the module is in *every state* $\sigma' \in \Sigma$. This intuition becomes precise when we consider multi-step executions of a module, depicted in Fig. 8: we pick some successor set Σ and then proceed for every possible $\sigma' \in \Sigma$.

Simulation. For modules $M_1, M_2 \in \mathsf{Module}(E)$, we define refinement as the simulation $(\preceq_{\mathrm{co}})$:

$$M_1 \preceq M_2 \triangleq (M_1, \sigma^0_{M_1}) \preceq_{\mathrm{co}} (M_2, \sigma^0_{M_2})$$

$$(M_1, \sigma_1) \preceq_{\mathrm{co}} (M_2, \sigma_2) \triangleq_{\mathrm{coind}} \forall e, \Sigma_1.\ \sigma_1 \xrightarrow{\alpha}_{M_1} \Sigma_1 \Rightarrow \exists \Sigma_2.\ \sigma_2 \xrightarrow{\alpha}{}^*_{M_2} \Sigma_2 \wedge \forall \sigma_2' \in \Sigma_2.\ \exists \sigma_1' \in \Sigma_1.\ (M_1, \sigma_1') \preceq_{\mathrm{co}} (M_2, \sigma_2')$$

Here, $(\preceq_{\mathrm{co}})$ is a coinductive simulation inspired by Alur et al. [1998] and Fritz and Wilke [2005]. For every step of the implementation $\sigma_1 \xrightarrow{\alpha}_{M_1} \Sigma_1$ with label α, the simulation demands a corresponding multi-step of the specification $\sigma_2 \xrightarrow{\alpha}{}^*_{M_2} \Sigma_2$ (where $\sigma \xrightarrow{\alpha}{}^*_M \Sigma$ is defined as $\sigma \xrightarrow{\alpha ::_? \mathsf{nil}}{}^*_M \Sigma$). This part of the definition is standard for a simulation with demonic non-determinism. Then the sides flip, and for every possible *successor state of the specification* $\sigma_2' \in \Sigma_2$, the simulation demands a corresponding state $\sigma_1' \in \Sigma_1$ such that σ_1' and σ_2' are in the simulation again. This second part is only present in simulations with angelic non-determinism. The simulation is a preorder:

Lemma 3.1. *$M_1 \preceq M_2$ is reflexive and transitive.*

Simulation vs. trace refinement. The reader might wonder why we center our reasoning around a simulation instead of another form of refinement (*e.g.*, a contextual refinement or a trace refinement). In DimSum, we work with a simulation, because simulations are sensitive to branching (*i.e.*, the order of visible events and non-deterministic choices) and branching sensitivity is crucial to implement linking as event-synchronization, as we will see in §3.2. Fortunately, the simulation $M_1 \preceq M_2$ strikes exactly the right balance: it is large enough to contain our desired examples (*e.g.*, the compiler passes in §5 and the coroutine linking operator in §4.3), it is compositional enough to be compatible with the operators of DimSum (see §3.3), and it is small enough to imply a traditional whole-program trace refinement. More specifically, we define whole-program trace refinement as $M_1 \sqsubseteq_{\mathcal{T}} M_2 \triangleq \mathcal{T}(M_1) \subseteq \mathcal{T}(M_2)$ where $\mathcal{T}(M) \triangleq \left\{\overline{e} \,\middle|\, \sigma^0 \xrightarrow{\overline{e}}{}^* S_M\right\}$ and, as to be expected, we obtain:

Theorem 3.2. *If $M_1 \preceq M_2$, then $M_1 \sqsubseteq_{\mathcal{T}} M_2$.*

$$\textsc{sim-vis}\ \frac{[\![p]\!]_s \preceq [\![p']\!]_s}{[\![\mathsf{vis}(e);p]\!]_s \preceq [\![\mathsf{vis}(e);p']\!]_s} \qquad \textsc{sim-ex-r}\ \frac{\exists y \in T.\, M \preceq [\![p(y)]\!]_s}{M \preceq [\![\exists x : T; p(x)]\!]_s} \qquad \textsc{sim-ex-l}\ \frac{\forall y \in T.\, [\![p(y)]\!]_s \preceq M}{[\![\exists x : T; p(x)]\!]_s \preceq M}$$

$$\textsc{sim-all-r}\ \frac{\forall y \in T.\, M \preceq [\![p(y)]\!]_s}{M \preceq [\![\forall x : T; p(x)]\!]_s} \qquad \textsc{sim-all-l}\ \frac{\exists y \in T.\, [\![p(y)]\!]_s \preceq M}{[\![\forall x : T; p(x)]\!]_s \preceq M}$$

Fig. 9. Derived quantifier elimination/introduction rules for Spec-programs.

3.2 Angelic and Demonic Non-Determinism

As we have discussed in §2.3 and §3.1, modules in DimSum have two kinds of non-determinism: *demonic* and *angelic non-determinism*. To understand when we want to use one vs. the other and how they affect proofs of the simulation $M_1 \preceq M_2$, we discuss them in the context of a concrete example: the specification language Spec. As mentioned in §2.1, we have so far only discussed a fragment of Spec. Formally, the full language is defined coinductively as follows:

$$\mathsf{Spec}(E) \ni p ::=_{\mathsf{coind}} \mathsf{vis}(e); p \mid \exists x : T; p(x) \mid \forall x : T; p(x) \qquad (e \in E)$$

As before, $\mathsf{vis}(e); p$ emits a visible event $e \in E$. The program $\exists x : T; p(x)$ uses demonic non-determinism—think "∃" reminiscent of the devil's trident Ψ—to choose some $x : T$ and then proceed as p. The program $\forall x : T; p(x)$ uses angelic non-determinism—think "∀" for an inverted A for angel—to assume a choice $x : T$ and then proceed as p. Besides the analogy, as we will see shortly, the symbols "∀" and "∃" also have a more literal reading as quantifiers in the context of the simulation.

Formally, the module semantics of a program $p \in \mathsf{Spec}(E)$ is a module $[\![p]\!]_s \triangleq (\mathsf{Spec}(E), \rightarrow_s, p)$ where programs execute according to the following transition system:

$$(\mathsf{vis}(e); p) \xrightarrow{e}_s \{p\} \qquad (\exists x : T; p(x)) \xrightarrow{\tau}_s \{p(y)\}\ (\text{for } y \in T) \qquad (\forall x : T; p(x)) \xrightarrow{\tau}_s \{p(y) \mid y \in T\}$$

We embed constructs of our meta theory (*e.g.*, if ϕ then p_1 else p_2) into Spec[8] and, hence, we can derive the remaining constructs of Spec from §2.1 using the three primitives:

$$\mathsf{any} \triangleq \mathsf{ub} \triangleq_{\mathsf{coind}} \forall x : \emptyset; \mathsf{ub} \qquad \mathsf{assume}(\phi); p \triangleq \text{if } \phi \text{ then } p \text{ else ub}$$
$$\mathsf{nb} \triangleq_{\mathsf{coind}} \exists x : \emptyset; \mathsf{nb} \qquad \mathsf{assert}(\phi); p \triangleq \text{if } \phi \text{ then } p \text{ else nb}$$

The specification any means the program can have *any behavior* or, in other words, the behavior is not defined (*i.e.*, ub). Formally, we can represent this behavior as an angelic choice over the empty set, because $M \preceq [\![\forall x : \emptyset; p]\!]_s$ for any M (*cf.* the definition of $(\preceq)$). The specification nb means the program has finished executing or, in other words, the program has *no behavior* anymore. Formally, we can represent termination as a demonic choice over the empty set: there is no "next" state that the program can step to. We can then derive the constructs $\mathsf{assume}(\phi); p$ and $\mathsf{assert}(\phi); p$.[9]

Non-deterministic choices as quantifiers. As mentioned above, the notation for demonic and angelic choice in Spec is no accident: the different forms of non-determinism have a reading as logical connectives in a simulation. The interactions of the two kinds of non-determinism with simulation are depicted in Fig. 9. If we read simulation "$\preceq$" as a form of implication "$\Rightarrow$", then the proof rules for the two quantifiers correspond to the introduction and elimination rules for universal and existential quantification of first-order logic. For example, existential quantification

[8] Spec is a shallow embedding in Coq and therefore inherits its rich collection of datatypes (*e.g.*, $\mathbb{N}$, $\mathsf{list}(T)$, *etc.*) and functions.
[9] The Coq development uses the (classically) equivalent definitions $\mathsf{assume}(\phi); p \triangleq \forall \_ : \phi; p$ and $\mathsf{assert}(\phi); p \triangleq \exists \_ : \phi; p$.

SIM-ALL-ALL-COMM
$[\![\forall x; \forall y; p(x,y)]\!]_s \preceq [\![\forall y; \forall x; p(x,y)]\!]_s$

SIM-EX-EX-COMM
$[\![\exists x; \exists y; p(x,y)]\!]_s \preceq [\![\exists y; \exists x; p(x,y)]\!]_s$

SIM-EX-ALL-COMM
$[\![\exists x; \forall y; p(x,y)]\!]_s \preceq [\![\forall y; \exists x; p(x,y)]\!]_s$

NO-SIM-ALL-EX-COMM
$[\![\forall y; \exists x; p(x,y)]\!]_s \not\preceq [\![\exists x; \forall y; p(x,y)]\!]_s$

SIM-EX-VIS-COMM
$[\![\exists x; \mathsf{vis}(e); p(x)]\!]_s \preceq [\![\mathsf{vis}(e); \exists x; p(x)]\!]_s$

NO-SIM-VIS-EX-COMM
$[\![\mathsf{vis}(e); \exists x; p(x)]\!]_s \not\preceq [\![\exists x; \mathsf{vis}(e); p(x)]\!]_s$

SIM-VIS-ALL-COMM
$[\![\mathsf{vis}(e); \forall x; p(x)]\!]_s \preceq [\![\forall x; \mathsf{vis}(e); p(x)]\!]_s$

NO-SIM-ALL-VIS-COMM
$[\![\forall x; \mathsf{vis}(e); p(x)]\!]_s \not\preceq [\![\mathsf{vis}(e); \forall x; p]\!]_s$

Fig. 10. Admissible and inadmissible quantifier commuting principles for Spec-programs.

$\exists x : T; p$ on the left side (SIM-EX-L) means we need to consider all possible choices of x, whereas existential quantification on the right side (SIM-EX-R) means we need to choose x. Furthermore, the quantifiers validate and invalidate all the usual quantifier commuting principles shown in Fig. 10.

The reading of non-deterministic choices as quantifiers generalizes beyond Spec. When we prove a simulation $M_1 \preceq M_2$, then demonic non-determinism in M_1 means we need to consider all possible choices; in M_2 it means we need to provide a particular choice. For angelic non-determinism, the rules are flipped. In M_1, we need to provide a particular choice; in M_2, we need to consider all possible choices. For example, in §2.3, we have discussed angelic non-determinism in the wrapper $\lceil M \rceil_{r \rightleftharpoons a}$. Recall that angelic non-determinism in this case meant that the wrapper can assume "the right choice" is provided to it by the angel. When we prove the simulation $\lceil M_1 \rceil_{r \rightleftharpoons a} \oplus_a \lceil M_2 \rceil_{r \rightleftharpoons a} \preceq \lceil M_1 \oplus_r M_2 \rceil_{r \rightleftharpoons a}$, then we have to slip into the role of the angel: for calls from M_1 to M_2, we obtain the "right choice" of, *e.g.*, the memory m through demonic non-determinism in M_1 (think "$\exists m$") and then we pass it on through angelic non-determinism in M_2 (think "$\forall m$").

Branching-sensitivity and linking. What we have not discussed so far is the interaction of visible events and non-deterministic choices. As it turns out, it is crucial that the simulation $M_1 \preceq M_2$ preserves the order of visible events and certain choices. More specifically, the rules SIM-EX-VIS-COMM and SIM-VIS-ALL-COMM of Fig. 10 are admissible whereas NO-SIM-VIS-EX-COMM and NO-SIM-ALL-VIS-COMM are not. Intuitively, the reason is that *linking* can "inline" an entire module in the place of a visible event e, so whenever we commute a quantifier over e, we are effectively commuting it over all the choices made "on the other side" of e.

To illustrate this point, let us consider a concrete example: we will show that if one admits the commuting forbidden by NO-SIM-ALL-VIS-COMM, then the simulation is trivial in the sense that $[\![p_1]\!]_s \preceq [\![p_2]\!]_s$ for any specifications p_1 and p_2. For this example, we define $p_L \triangleq \forall x : \emptyset; \mathsf{vis}(A!); \mathsf{any}$ and $p_R \triangleq \mathsf{vis}(A?); p_2$ and consider what happens when we link them together with a suitably defined linking operation (*i.e.*, one matching $A!$ with $A?$). Using the forbidden commuting, we show:

$$[\![p_1]\!]_s \preceq [\![p_L \oplus p_R]\!]_s \preceq [\![p_2]\!]_s$$

For the first part, $[\![p_1]\!]_s \preceq [\![p_L \oplus p_R]\!]_s$, it suffices to observe that $p_L \oplus p_R$ can be implemented by *any* program p, because it starts with an angelic choice over an empty set. That is, suppose the linked program $p_L \oplus p_R$ starts executing on the left side. Then we are given $x \in \emptyset$ in the proof of $[\![p_i]\!]_s \preceq [\![p_L \oplus p_R]\!]_s$ (by SIM-ALL-R) and we are done. For the second part, $[\![p_L \oplus p_R]\!]_s \preceq [\![p_2]\!]_s$ we use the commuting rule NO-SIM-ALL-VIS-COMM. With the commuting rule, it suffices to show $[\![(\mathsf{vis}(A!); \forall x : \emptyset; \mathsf{any}) \oplus (\mathsf{vis}(A?); p_2)]\!]_s \preceq [\![p_2]\!]_s$, which follows from executing the module: we start on the left, synchronize on A and continue execution on the right, and then continue with p_2.

$$\textsc{product-compat}\ \frac{M_1 \preceq M_1' \qquad M_2 \preceq M_2'}{M_1 \times M_2 \preceq M_1' \times M_2'} \qquad \textsc{filter-compat}\ \frac{M_1 \preceq M_1'}{M_1 \setminus M_2 \preceq M_1' \setminus M_2} \qquad \textsc{link-compat}\ \frac{M_1 \preceq M_1' \qquad M_2 \preceq M_2'}{M_1 \oplus_X M_2 \preceq M_1' \oplus_X M_2'} \qquad \textsc{wrapper-compat}\ \frac{M \preceq M'}{\lceil M \rceil_X \preceq \lceil M' \rceil_X}$$

Fig. 11. Compositional combinator reasoning principles.

In summary, angelic and demonic non-determinism allow modules to express universal and existential quantification, which enables the local encoding of assumptions about their environment and guarantees about their own behavior (as used by the $\lceil \mathsf{M} \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}$ wrapper). To ensure that the semantics of operations like linking can be meaningfully expressed as compositions of modules, DimSum relies on a branch-sensitive simulation that carefully controls the commutation of visible events and non-deterministic choices.

3.3 Combinators

One of the strong suits of DimSum is that it comes with a compositional set of language-agnostic combinators. Concretely, DimSum provides out-of-the-box a combinator for the product $M_1 \times M_2$ of two modules M_1 and M_2, one for filtering $M_1 \setminus M_2$, one for linking $M_1 \oplus_X M_2$, and one for stateful wrappers $\lceil M \rceil_X$. The combinators we have encountered so far— $\oplus_{\mathsf{r}}$, $\oplus_{\mathsf{a}}$, and $\lceil \cdot \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}$—are all language-specific instantiations of these generic combinators (see §4 and §5). To allow for the compositional reasoning we aim for in DimSum, all of them need to be compatible with simulation, *i.e.*, they need to be monotone with respect to $\preceq$, as asserted, for example, by ASM-LINK-HORIZONTAL. The main benefit of expressing the language-specific combinators as instances of the generic combinators is that the desired compatibility properties—shown in Fig. 11—can be proven once and for all for the generic combinators.[10] In the following, we will discuss the definition of the product combinator $M_1 \times M_2$ in detail and, for the sake of brevity, only describe the functionality of the others (see the appendix [Sammler et al. 2023a, §A] for details).

Product. The product combinator $M_1 \times M_2 \triangleq (\{\mathsf{E}, \mathsf{L}, \mathsf{R}\} \times S_{M_1} \times S_{M_2}, \rightarrow_\times, (\mathsf{E}, \sigma^0_{M_1}, \sigma^0_{M_2}))$ builds the product of two modules M_1 and M_2. It is inspired by parallel composition in process calculi such as CSP [Hoare 1978; Roscoe 2010] and CCS [Milner et al. 1992; Milner 1999], but restricted to a particular form of scheduling (depicted in Fig. 12): At any point in time, either the environment, the left module M_1, or the right module M_2 is executing. We store whose turn it currently is in a flag $d \in \mathsf{D} \triangleq \{\mathsf{E}, \mathsf{L}, \mathsf{R}\}$ as part of the state of the module (alongside the state of the two modules M_1 and M_2). If the executing party is one of M_1 or M_2 and executes a silent step (PRODUCT-STEP-L-SILENT and PRODUCT-STEP-R-SILENT), then it remains their turn. We only switch turns once the module emits a visible event (PRODUCT-STEP-L and PRODUCT-STEP-R). Whenever we switch, the next turn d is chosen using demonic non-determinism. If it is currently the environment's turn, then we non-deterministically choose d for the next step (PRODUCT-STEP-ENV).

The events of the module $e_\times \triangleq \mathsf{left}(e, d) \mid \mathsf{right}(e, d) \mid \mathsf{env}(d)$ expose the scheduling choice of the product combinator (*i.e.*, who is currently executing). We can exploit this information in other combinators to construct more deterministic schedules. For example, the linking combinator $M_1 \oplus_X M_2$ can filter out certain scheduling choices of $M_1 \times M_2$ and thereby enforce structured communication between M_1 and M_2 (as seen in §2.2).

[10] Technically, these compatibility properties only hold for modules that do not perform (non-trivial) angelic choices on steps with visible events. This requirement is trivial to satisfy by moving the angelic choice into a separate silent step.

$$\textsc{product-step-l}\quad \frac{\sigma_1 \xrightarrow{e} \Sigma}{(\mathsf{L},\sigma_1,\sigma_2) \xrightarrow{\mathsf{left}(e,d)}_{\times} \left\{(d,\sigma_1',\sigma_2) \,\middle|\, \sigma_1' \in \Sigma\right\}} \qquad \textsc{product-step-l-silent}\quad \frac{\sigma_1 \xrightarrow{\tau} \Sigma}{(\mathsf{L},\sigma_1,\sigma_2) \xrightarrow{\tau}_{\times} \left\{(\mathsf{L},\sigma_1',\sigma_2) \,\middle|\, \sigma_1' \in \Sigma\right\}}$$

$$\textsc{product-step-r}\quad \frac{\sigma_2 \xrightarrow{e} \Sigma}{(\mathsf{R},\sigma_1,\sigma_2) \xrightarrow{\mathsf{right}(e,d)}_{\times} \left\{(d,\sigma_1,\sigma_2') \,\middle|\, \sigma_2' \in \Sigma\right\}} \qquad \textsc{product-step-r-silent}\quad \frac{\sigma_2 \xrightarrow{\tau} \Sigma}{(\mathsf{R},\sigma_1,\sigma_2) \xrightarrow{\tau}_{\times} \left\{(\mathsf{R},\sigma_1,\sigma_2') \,\middle|\, \sigma_2' \in \Sigma\right\}}$$

$$\textsc{product-step-env}\quad (\mathsf{E},\sigma_1,\sigma_2) \xrightarrow{\mathsf{env}(d)}_{\times} \{(d,\sigma_1,\sigma_2)\}$$

Fig. 12. Definition of $\rightarrow_{\times}$.

Filter. For a module $M \in \mathsf{Module}(E_1)$, the filter combinator $M \setminus M' \in \mathsf{Module}(E_2)$ transforms the events of the left module. That is, intuitively, one can think of the filter M' as a relation "$e_1 \sim e_2 \subseteq E_1 \times E_2$" that is used to turn events $e_1 \in E_1$ into events $e_2 \in E_2$ and vice versa. In practice, expressing the filter M' in terms of *a relation* is too restrictive, because we sometimes want (1) to carry state in between the event translations, (2) map a single event to multiple events, and (3) use angelic and demonic non-determinism to control how the events are filtered. Thus, in DimSum, we go beyond a relation "$e_1 \sim e_2$" and instead use a *filter module* M'. This is similar to the notion of transducers in automata theory. The idea is that when M emits an event $e_1 \in E_1$, control is passed to the module M', which will then execute and emit one or more events to the environment. To be precise, the filter module M' communicates using the events:

$$e_{\setminus} ::= \mathsf{FromInner}(e_1 : E_1) \mid \mathsf{ToInner}(e_1 : \mathsf{option}(E_1)) \mid \mathsf{ToEnv}(e_2 : E_2) \mid \mathsf{FromEnv}(e_2 : E_2)$$

$\mathsf{FromInner}(e_1)$ means that M' is willing to accept e_1 from M, $\mathsf{ToInner}(e_1)$ means that M' wants to return control to the module M, optionally sending it the event e_1, $\mathsf{ToEnv}(e_2)$ means that M' wants to send e_2 to the environment, and $\mathsf{FromEnv}(e_2)$ means that M' is willing to accept e_2 from the environment. Throughout all of these interactions, the filter M' can maintain some internal state, since it itself is a module (*i.e.*, state transition system).[11] We will see an instance of a filter module below when we discuss the wrapper combinator.

Linking. To express semantic linking in a *language-generic way*, the linking combinator $M_1 \oplus_X M_2$ works as follows. The modules M_1 and M_2 emit tagged events $e_{?!} \in E_{?!} \triangleq E \times \{?, !\}$ such as **Jump?(r, m)** or **Jump!(r, m)**, where the tag $t \in \{?, !\}$ indicates whether the event is incoming (?) or outgoing (!). It is then the job of the linking operator $\oplus_X$ to flip the event $e_{?!}$ or replace the event $e_{?!}$ with a different event $e'_{?!}$ (*e.g.*, for the coroutine linking operator in §4.3 calls become returns). Technically we define $M_1 \oplus_X M_2 \triangleq (M_1 \times M_2) \setminus \mathsf{link}_X$: the non-deterministic scheduling of $M_1 \times M_2$ is filtered by link_X, which discards out all the interleavings that are "nonsensical". For example, if M_1 wants to "jump" to the environment, then link_X filters out the interleavings of $M_1 \times M_2$ where the next turn is L (for M_1) or R (for M_2).

The parameter $X = (S, \rightsquigarrow, s^0)$ determines how the events are linked. It consists of a set of linking-internal states S, an initial state $s^0 \in S$, and a relation $\rightsquigarrow \subseteq (\mathsf{D} \times S \times E) \times ((\mathsf{D} \times S \times E) \cup \{\lightning\})$ describing how events should be translated. Concretely, $(d, s, e) \rightsquigarrow (d', s', e')$ means the untagged

[11] Readers familiar with process algebra can think of the filter combinator $M \setminus M'$ as the process $(M \parallel M') \setminus E_1$ where the module M' accepts the events from M and emits events $e_2 \in E_2$.

event $e \in E$ coming from direction d should go to d' as the event e'. Behind the scenes, the linking operator then adds the right tag $t \in \{?, !\}$ to e', depending on whether e is part of an incoming or outgoing event. It is also possible that the event cannot be linked $(d, s, e) \rightsquigarrow \lightning$, in which case the linking $M_1 \oplus_X M_2$ has undefined behavior. The linking-internal states S are a form of private state that the linking operator can use to remember information across invocations (*e.g.*, a syscall triggered by the left module should return to the left module). We will discuss concrete instantiations of the linking relation $\rightsquigarrow$ in §4.

(Kripke) wrappers. For $M \in \mathsf{Module}(E_1)$, the wrapper $\lceil M \rceil_X \triangleq M \setminus \mathsf{wrap}_X \in \mathsf{Module}(E_2)$ translates events between languages with events E_1 and E_2 (*e.g.*, between Rec and **Asm** in the case of $\lceil \cdot \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}$). The combinator is a special case of filtering where the filter wrap_X encodes a particular event translation. The parameter $X = (\mathcal{L}, \rightharpoonup, \leftharpoonup)$ contains a separation logic $\mathcal{L}$ (explained below) and a pair of Kripke relations ($\rightharpoonup$ and $\leftharpoonup$) where $e_1 \rightharpoonup e_2$ controls the translation E_1 to E_2 and $e_1 \leftharpoonup e_2$ controls the translation E_2 to E_1. In both directions, we use non-determinism in the filter wrap_X to pick "the right" corresponding events. For $e_2 \in E_2$ arriving from the environment, the filter *angelically* chooses an event e_1 such that $e_1 \leftharpoonup e_2$. For $e_1 \in E_1$ originating from the module M, the filter *demonically* chooses an event e_2 such that $e_1 \rightharpoonup e_2$.

The relations $\rightharpoonup$ and $\leftharpoonup$ are Kripke relations in the sense that they maintain state between events. Instead of explicitly indexing these relations with a "possible world" (as sketched in §2.3), we define them in separation logic. That is, their type is $\rightharpoonup, \leftharpoonup : E_1 \times E_2 \to \mathit{Prop}_{\mathcal{L}}$ where $\mathit{Prop}_{\mathcal{L}}$ denotes the type of propositions in the separation logic $\mathcal{L}$ (one component of X). The separation logic $\mathcal{L}$ determines which "resources" the relations $\rightharpoonup$ and $\leftharpoonup$ can refer to (*e.g.*, a Rec or **Asm** heap). We always use separation logics $\mathcal{L}$ that are instances of the separation logic framework Iris [Jung et al. 2015, 2018].[12] We will see a concrete choice of $\mathcal{L}$ and the relations $\rightharpoonup$ and $\leftharpoonup$ in §5.

To understand how the wrapper works, it is instructive to discuss the definition of the filter module wrap_X. It is given by $\mathsf{wrap}_X \triangleq [\![\mathsf{wrap}(\mathsf{True})]\!]_{\mathsf{s}}$ where wrap coordinates the exchange between the wrapped module M and its environment. The argument of wrap is a separation logic proposition keeping track of the "resources" that the inner module owns privately (*i.e.*, that are not shared with the environment). The definition of wrap is given by:

$$
\begin{aligned}
&\mathsf{wrap}(P_1) \triangleq_{\mathsf{coind}} \\
&\quad \exists e_2; \mathsf{vis}(\mathsf{FromEnv}(e_2)); \forall e_1, P_2; \mathsf{assume}(\mathsf{sat}(P_1 * P_2 * e_1 \leftharpoonup e_2)); \mathsf{vis}(\mathsf{ToInner}(e_1)); \\
&\quad \exists e_1'; \mathsf{vis}(\mathsf{FromInner}(e_1')); \exists e_2', P_1'; \mathsf{assert}(\mathsf{sat}(P_1' * P_2 * e_1' \rightharpoonup e_2')); \mathsf{vis}(\mathsf{ToEnv}(e_2')); \mathsf{wrap}(P_1')
\end{aligned}
$$

Initially, the filter accepts any incoming event e_2 from the environment (with $\mathsf{FromEnv}(e_2)$). It then assumes it is given *angelically* the corresponding event e_1 for the inner module, which it sends to the module (with $\mathsf{ToInner}(e_1)$). Afterwards, the filter module accepts any response event e_1' from the inner module (with $\mathsf{FromInner}(e_1')$). It then *demonically* chooses the corresponding event e_2' to send to the environment (with $\mathsf{ToEnv}(e_2')$).

In the exchange between the wrapped module M and the environment, separation logic propositions are used to "divide up" the shared state (*e.g.*, the locations in the heap). The wrapped module exclusively owns some resources P_1 which the environment may not change, and it can update them to P_1' during the exchange. The environment exclusively owns some resources P_2 which the wrapped module must preserve while updating its own state.[13] Finally, during every exchange

[12] In fact, readers familiar with Iris can think of the separation logic $\mathcal{L}$ as $\mathit{UPred}(R)$, the separation logic of uniform predicates over the resource algebra R, where each instance of the wrapper combinator chooses its own resource algebra R.

[13] The structure of this exchange follows Iris's frame-preserving update modality [Jung et al. 2018]: Initially, the module assumes some decomposition of the shared resources, and then it makes sure to only update the exclusively owned resources P_1 to resources P_1' that remain compatible with the environment resources.

$$\begin{aligned}
\mathbf{Library} \ni \mathbf{A} &\triangleq \mathbb{Z} \xrightarrow{\text{fin}} \mathbf{Instr} \\
\mathbf{Instr} \ni \mathbf{c} &::= \mathsf{ret} \mid \mathsf{mov}\ \mathbf{x},\ \mathbf{o} \mid \mathsf{add}\ \mathbf{x}_1,\ \mathbf{x}_2,\ \mathbf{o} \mid \mathsf{mul}\ \mathbf{x}_1,\ \mathbf{x}_2,\ \mathbf{o} \mid \mathsf{sle}\ \mathbf{x}_1,\ \mathbf{x}_2,\ \mathbf{o} \mid \mathsf{syscall} \\
&\quad \mid \mathsf{ldr}\ \mathbf{x}_1, [\mathbf{x}_2 + i] \mid \mathsf{str}\ \mathbf{x}_1, [\mathbf{x}_2 + i] \mid \mathsf{jmp}\ \mathbf{o} \mid \mathsf{beq}\ \mathbf{o}_1,\ \mathbf{x},\ \mathbf{o}_2 \mid \cdots \\
\mathbf{Operand} \ni \mathbf{o} &::= \mathbf{x} : \mathbf{RegisterName} \mid i : \mathbb{Z} \\
\mathbf{Execution\ State} \ni \mathbf{I} &::= \mathbf{Wait} \mid \mathbf{Run}(\mathbf{r}, \mathbf{m}) \mid \mathbf{WaitSyscall}(\mathbf{r}) \mid \mathbf{Halted}
\end{aligned}$$

ASM-INCOMING

$$\frac{\mathbf{r}(\mathbf{pc}) = \mathbf{a} \qquad \mathbf{a} \in |\mathbf{A}|}{(\mathbf{Wait}, \mathbf{A}) \xrightarrow{\mathbf{Jump?}(\mathbf{r},\mathbf{m})}_{\mathbf{a}} \{(\mathbf{Run}(\mathbf{r}, \mathbf{m}), \mathbf{A})\}}$$

ASM-JUMP-INTERNAL

$$\frac{\mathbf{r}(\mathbf{pc}) = \mathbf{a} \qquad \mathbf{A}(\mathbf{a}) = \mathsf{jmp}\ \mathbf{v} \qquad \mathbf{v} \in |\mathbf{A}|}{(\mathbf{Run}(\mathbf{r}, \mathbf{m}), \mathbf{A}) \xrightarrow{\tau}_{\mathbf{a}} \{(\mathbf{Run}(\mathbf{r}[\mathbf{pc} \mapsto \mathbf{v}], \mathbf{m}), \mathbf{A})\}}$$

ASM-JUMP-EXTERNAL

$$\frac{\mathbf{r}(\mathbf{pc}) = \mathbf{a} \qquad \mathbf{A}(\mathbf{a}) = \mathsf{jmp}\ \mathbf{v} \qquad \mathbf{v} \notin |\mathbf{A}|}{(\mathbf{Run}(\mathbf{r}, \mathbf{m}), \mathbf{A}) \xrightarrow{\mathbf{Jump!}(\mathbf{r}[\mathbf{pc} \mapsto \mathbf{v}], \mathbf{m})}_{\mathbf{a}} \{(\mathbf{Wait}, \mathbf{A})\}}$$

Fig. 13. Grammar and excerpt of the operational semantics of Asm.

some parts of the state can be shared between the environment and the wrapped module using the separation logic relations $e_1 \rightharpoonup e_2$ and $e_1 \leftharpoonup e_2$ (see §5). The proposition $\mathrm{sat}(P)$ (read P is satisfiable) here means that there is some valid underlying resource (*e.g.*, a heap) for which P holds.

4 INSTANTIATIONS OF THE FRAMEWORK

4.1 The Language Asm

The language Asm is an idealized assembly language with instructions for arithmetic, jumps, memory accesses, and syscalls (depicted in Fig. 13).[14] The libraries A of Asm are finite maps from addresses to instructions. The set of their instruction addresses is defined as $|\mathbf{A}| = \mathrm{dom}\,\mathbf{A}$.

Module semantics. The semantics of an Asm library A is the module $[\![\mathbf{A}]\!]_{\mathbf{a}}$. We write $(\rightarrow_{\mathbf{a}})$ for the transition system (excerpt shown in Fig. 13). The states of the module are of the form $\sigma = (\mathbf{I}, \mathbf{A})$ where I is the current *execution state*, and the initial state is $(\mathbf{Wait}, \mathbf{A})$. Conceptually, four different execution states are possible during the execution of A: executing ($\mathbf{Run}(\mathbf{r}, \mathbf{m})$), waiting for incoming jumps (Wait), waiting for a syscall to return ($\mathbf{WaitSyscall}(\mathbf{r})$, where r preserves the registers across the syscall), or finished executing (Halted). To explain the transitions, we discuss three cases. Initially, the module is waiting (Wait) and can accept any incoming jump with arbitrary memory and registers (see ASM-INCOMING). After accepting the jump ($\mathbf{Run}(\mathbf{r}, \mathbf{m})$), the module executes the instructions of A, updating the current register assignment r and memory m (not shown in the figure). When the module reaches a jump $\mathsf{jmp}\ \mathbf{v}$ (or jumps to an instruction in another way), one of two things happens: Either the destination v is in the address range of A and we continue executing in the module (see ASM-JUMP-INTERNAL), or the destination v is outside of the address range of A. In the latter case, the module emits $\mathbf{Jump!}(\mathbf{r}[\mathbf{pc} \mapsto \mathbf{v}], \mathbf{m})$ and returns to the Wait state. Syscalls execute analogously to jumps (with WaitSyscall), and a library can finish executing (Halted).

Linking. Syntactically, linking of two Asm libraries (*i.e.*, $\mathbf{A}_1 \cup_{\mathbf{a}} \mathbf{A}_2$) means merging the maps $\mathbf{A}_1$ and $\mathbf{A}_2$. In case of overlapping addresses, the conflict is resolved by using the instruction from

[14] Following Sammler et al. [2022], the Coq development defines the instructions depicted in Fig. 13 as compositions of micro-instructions. A description of this approach can be found in the appendix [Sammler et al. 2023a, §B].

ASM-LINK-JUMP
$$\frac{(d' = \mathsf{L} \wedge \mathbf{r(pc)} \in \mathbf{d_1}) \vee (d' = \mathsf{R} \wedge \mathbf{r(pc)} \in \mathbf{d_2}) \vee (d' = \mathsf{E} \wedge \mathbf{r(pc)} \notin \mathbf{d_1} \cup \mathbf{d_2}) \qquad d \neq d'}{(d, \mathsf{None}, \mathbf{Jump(r, m)}) \rightsquigarrow_{\mathbf{d_1},\mathbf{d_2}} (d', \mathsf{None}, \mathbf{Jump(r, m)})}$$

REC-LINK-CALL
$$\frac{(d' = \mathsf{L} \wedge \mathsf{f} \in \mathsf{d}_1) \vee (d' = \mathsf{R} \wedge \mathsf{f} \in \mathsf{d}_2) \vee (d' = \mathsf{E} \wedge \mathsf{f} \notin \mathsf{d}_1 \cup \mathsf{d}_2) \qquad d \neq d'}{(d, \overline{d_s}, \mathsf{Call}(\mathsf{f}, \overline{\mathsf{v}}, \mathsf{m})) \rightsquigarrow_{\mathsf{d}_1,\mathsf{d}_2} (d', d :: \overline{d_s}, \mathsf{Call}(\mathsf{f}, \overline{\mathsf{v}}, \mathsf{m}))}$$

REC-LINK-RET
$$\frac{d \neq d'}{(d, d' :: \overline{d_s}, \mathsf{Return}(\mathsf{v}, \mathsf{m})) \rightsquigarrow_{\mathsf{d}_1,\mathsf{d}_2} (d', \overline{d_s}, \mathsf{Return}(\mathsf{v}, \mathsf{m}))}$$

CORO-LINK-YIELD
$$\frac{(d = \mathsf{L} \wedge d' = \mathsf{R}) \vee (d = \mathsf{R} \wedge d' = \mathsf{L})}{(d, (d, \mathsf{None}), \mathsf{Call}(\mathsf{yield}, [\mathsf{v}], \mathsf{m})) \rightsquigarrow_{\mathsf{coro}}^{\mathsf{d}_1,\mathsf{d}_2} (d', (d', \mathsf{None}), \mathsf{Return}(\mathsf{v}, \mathsf{m}))}$$

Fig. 14. Excerpt of the semantic linking relations of **Asm** ($\rightsquigarrow$), Rec ($\rightsquigarrow$), and Rec with coroutines ($\rightsquigarrow_{\mathsf{coro}}$).

$$\begin{aligned}
\mathsf{Library} \ni \mathsf{R} &::= (\mathsf{fn}\ \mathsf{f}(\overline{x}) \triangleq \overline{\mathsf{local}\ y[n]}; \mathsf{e}), \mathsf{R} \mid \emptyset \\
\mathsf{Expr} \ni \mathsf{e} &::= \mathsf{v} \mid x \mid \mathsf{e}_1 \oplus \mathsf{e}_2 \mid \mathsf{let}\ x := \mathsf{e}_1\ \mathsf{in}\ \mathsf{e}_2 \mid \mathsf{if}\ \mathsf{e}_1\ \mathsf{then}\ \mathsf{e}_2\ \mathsf{else}\ \mathsf{e}_3 \mid \mathsf{e}_1(\overline{\mathsf{e}_2}) \mid \mathsf{!e} \mid \mathsf{e}_1 \leftarrow \mathsf{e}_2 \\
\mathsf{BinOp} \ni \oplus &::= + \mid < \mid == \mid \leq
\end{aligned}$$

Fig. 15. Grammar of Rec.

$\mathbf{A_1}$ (the choice of $\mathbf{A_1}$ over $\mathbf{A_2}$ is arbitrary). Semantically, linking is more interesting. If we link two **Asm** modules (*i.e.*, $\mathbf{M_1}\ {}^{\mathbf{d_1}}\!\oplus_{\mathbf{a}}^{\mathbf{d_2}}\ \mathbf{M_2}$), then we have to synchronize based on the jump events. To define $\mathbf{M_1}\ {}^{\mathbf{d_1}}\!\oplus_{\mathbf{a}}^{\mathbf{d_2}}\ \mathbf{M_2}$, we use the combinator $M_1 \oplus_X M_2$ from §3.3. In the case of **Asm**, we pick $X = (\mathsf{option}(\mathsf{D}), \rightsquigarrow_{\mathbf{d_1},\mathbf{d_2}}, \mathsf{None})$. For a jump event (see ASM-LINK-JUMP in Fig. 14), the linking operator resolves the destination d' based on the addresses in $\mathbf{d_1}$ and $\mathbf{d_2}$—jumps that are outside these addresses are passed on to the environment. Syscalls are propagated to the environment, as described in the appendix [Sammler et al. 2023a, §C].

4.2 The Language Rec

The language Rec is a simple, high-level language with arithmetic operations, let bindings, memory operations, conditionals, and (potentially recursive) function calls (depicted in Fig. 15). The libraries R of Rec are lists of function declarations. Each function declaration contains the name of the function f, the argument names $\overline{x}$, local variables $\overline{y}$ which are allocated in the memory, and a function body e. The set of function names $|\mathsf{R}|$ of a library R is defined as the names of the functions in the list R. The module semantics for Rec is given in the appendix [Sammler et al. 2023a, §D]. The syntactic linking $\mathsf{R}_1 \cup_{\mathsf{r}} \mathsf{R}_2$ merges the function declarations of both libraries (again giving precedence to the left side in case of conflict). Similar to **Asm**, the semantic linking $\mathbf{M_1}\ {}^{\mathbf{d_1}}\!\oplus_{\mathbf{r}}^{\mathbf{d_2}}\ \mathbf{M_2}$ is an instance of $(\oplus_X)$ where the linking relation is depicted in Fig. 14. The most interesting difference to **Asm** is that linking in Rec has to build up and then wind down a call-stack (through calls and returns), which is maintained as the internal state of ($\rightsquigarrow$).

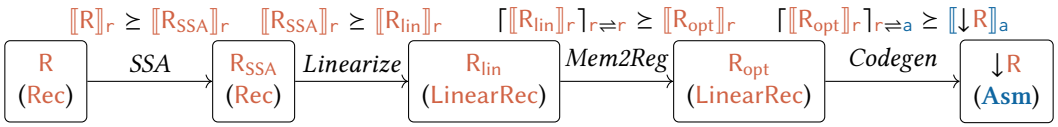


Fig. 16. Structure of our Rec to Asm compiler.

4.3 Coroutine Linking $M_1 \oplus_{\mathsf{coro}} M_2$

One of the strong suits of DimSum is that it allows multiple semantic linking operators for the same language. We showcase this using the coroutine linking operator $M_1 \oplus_{\mathsf{coro}} M_2$ from the example in §1.1. Similar to $(\oplus_a)$ and $(\oplus_r)$, the operator $M_1 \oplus_{\mathsf{coro}} M_2$ is an instance of $M_1 \oplus_X M_2$. The most interesting case of its transition relation $\rightsquigarrow_{\mathsf{coro}}$, CORO-LINK-YIELD, is depicted in Fig. 14.[15] Here, we can see that the linking operator links calls to yield from M_1 (resp. M_2) with returns from yield in M_2 (resp. M_1). This translation of calls to returns captures the intuitive behavior of the coroutine library **yield** (in Fig. 1) at the level of Rec—without mentioning the complex implementation of **yield** in **Asm**.

Verification of main and stream. We verify the example in Fig. 1, namely that it prints 0, then 1, and then returns 2. The proof strategy for this verification is analogous to §2.2:

$$\begin{aligned}\mathbf{coro} &\preceq \llbracket \mathbf{yield} \rrbracket_a \oplus_a \lceil \llbracket \mathsf{main} \rrbracket_r \rceil_{r\rightleftharpoons a} \oplus_a \lceil \llbracket \mathsf{stream} \rrbracket_r \rceil_{r\rightleftharpoons a} \oplus_a \llbracket \mathbf{print}_{\mathsf{spec}} \rrbracket_s \\ &\preceq \lceil \llbracket \mathsf{main} \rrbracket_r \oplus_{\mathsf{coro}} \llbracket \mathsf{stream} \rrbracket_r \rceil_{r\rightleftharpoons a} \oplus_a \llbracket \mathbf{print}_{\mathsf{spec}} \rrbracket_s \\ &\preceq \lceil \llbracket \mathsf{main}_{\mathsf{spec}} \rrbracket_s \rceil_{r\rightleftharpoons a} \oplus_a \llbracket \mathbf{print}_{\mathsf{spec}} \rrbracket_s \preceq \llbracket \mathbf{coro}_{\mathsf{spec}} \rrbracket_s\end{aligned}$$

Here **coro** denotes the compiled and syntactically linked **Asm**-program. The specifications $\mathbf{coro}_{\mathsf{spec}}$ and $\mathsf{main}_{\mathsf{spec}}$ are similar to the corresponding specifications in §2. The key steps of this proof are the second and third step which use the following rules:

CORO-LINK
$$\llbracket \mathbf{yield} \rrbracket_a \oplus_a \lceil M_1 \rceil_{r\rightleftharpoons a} \oplus_a \lceil M_2 \rceil_{r\rightleftharpoons a} \preceq \lceil M_1 \oplus_{\mathsf{coro}} M_2 \rceil_{r\rightleftharpoons a}$$

MAIN-CORO
$$\llbracket \mathsf{main} \rrbracket_r \oplus_{\mathsf{coro}} \llbracket \mathsf{stream} \rrbracket_r \preceq \llbracket \mathsf{main}_{\mathsf{spec}} \rrbracket_s$$

The lemma CORO-LINK is a generic lemma provided by the coroutine library that allows abstracting the **yield** library to the $M_1 \oplus_{\mathsf{coro}} M_2$. This lemma enables us to verify the composition of main and stream purely at the Rec-level (MAIN-CORO)—completely independently of **Asm**.

5 COMPILER

This section describes our compiler ↓R from Rec to **Asm**, and how we verify it in DimSum. The compiler has four passes, depicted in Fig. 16: The first pass, *SSA*, renames variables such that each variable is only assigned once, and the second pass, *Linearize*, converts the program into A-normal form. The A-normal form is expressed in an intermediate representation, LinearRec, which is a subset of Rec that only allows let-bindings and if-statements at the top-level and flattens all nested expressions. The third pass, *Mem2Reg*, is a non-trivial optimization pass that reduces memory consumption by turning local variables whose address is never observed into let-bindings (which can be compiled to registers subsequently). For example, it turns (the A-normal form of) $\mathsf{fn}\ \mathsf{f}(x) \triangleq \mathsf{local}\ y[1]; y \leftarrow x; !y + !y$ into $\mathsf{fn}\ \mathsf{f}(x) \triangleq \mathsf{let}\ y := 0\ \mathsf{in}\ \mathsf{let}\ y := x\ \mathsf{in}\ y + y$, because the *address* of y is never used. The final pass, *Codegen*, is a standard code-generation pass producing the **Asm** code: it takes care of register allocation (including spilling to the stack when necessary), allocating local variables on the stack, and adhering to the **Asm** calling convention.

[15] The full definition of $M_1 \oplus_{\mathsf{coro}} M_2$ can be found in the appendix [Sammler et al. 2023a, §F].

Compiler correctness. Let us turn to the correctness of the compiler (COMPILER-CORRECT in §2.2):[16]

THEOREM 5.1 (COMPILER CORRECTNESS). *If* $\downarrow \mathsf{R}$ *is defined, then* $[\![\downarrow \mathsf{R}]\!]_{\mathsf{a}} \preceq \lceil [\![\mathsf{R}]\!]_{\mathsf{r}} \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}$.

Intuitively, compiler correctness says that the compiled assembly code behaves like the original source program translated by the wrapper—*i.e.*, syntactic translation via the compiler refines semantic translation via the wrapper $\lceil \cdot \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}$. As we have seen in §2.2, COMPILER-CORRECT is a useful result that allows one to replace reasoning about the compiled assembly code with reasoning about the source program. The $\lceil \cdot \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}$ wrapper is defined using the $\lceil \cdot \rceil_X$ combinator from §3.3 and can be found in the appendix [Sammler et al. 2023a, §E].

The compiler correctness result is proven by composing refinements for the individual passes (the refinements are shown in Fig. 16 above each corresponding pass):

$$\lceil [\![\mathsf{R}]\!]_{\mathsf{r}} \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}} \succeq \lceil [\![\mathsf{R}_{\mathsf{SSA}}]\!]_{\mathsf{r}} \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}} \succeq \lceil [\![\mathsf{R}_{\mathsf{lin}}]\!]_{\mathsf{r}} \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}} \succeq \lceil \lceil [\![\mathsf{R}_{\mathsf{lin}}]\!]_{\mathsf{r}} \rceil_{\mathsf{r} \rightleftharpoons \mathsf{r}} \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}} \succeq \lceil [\![\mathsf{R}_{\mathsf{opt}}]\!]_{\mathsf{r}} \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}} \succeq [\![\downarrow \mathsf{R}]\!]_{\mathsf{a}}$$

The refinements for the *SSA* and *Linearize* passes are straightforward Rec refinements, and the *Codegen* pass uses the $\lceil \cdot \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}$ wrapper to translate between Rec and **Asm**. The pass *Mem2Reg* is special, however, in that it introduces an additional wrapper $\lceil \cdot \rceil_{\mathsf{r} \rightleftharpoons \mathsf{r}}$. To understand what this wrapper does and why we have to introduce it, let us first take a step back and consider how DimSum determines which program transformations are considered semantics-preserving.

Semantics-preserving program transformations. There are two classes of program transformations that are semantics-preserving in DimSum.

The first class of semantics-preserving program transformations *does not* change observable parts of the events. This can be expressed by proving that the transformed program refines the original, *i.e.*, $[\![\mathsf{R}_1]\!]_{\mathsf{r}} \preceq [\![\mathsf{R}_2]\!]_{\mathsf{r}}$ where R_1 is the transformed program and R_2 the original. This refinement expresses that the transformed program R_1 must emit the same events as the original R_2, because the definition of $(\preceq)$ (in §3.1) enforces that the events of R_1 and R_2 match exactly. Thus, by selecting which information about the program state to include in its events, a language effectively determines this first class of semantics-preserving program transformations. For example, Rec includes values v and heaps m in its events and thus program transformations can change the syntactic structure of a program—since the program structure is not part of the events—but they cannot alter return values or the heap that is shared across function calls (unless they fall into the second class). In the compiler, the *SSA* and *Linearize* passes fall into this first class of transformations (see their correctness statements in Fig. 16).

The second class of semantics-preserving program transformations *does* change observable parts of the events. For example, the *Mem2Reg* transformation falls into this category, because it alters memory in such a way that it becomes impossible to align the events of the transformed program $\mathsf{R}_{\mathsf{opt}}$ with the original $\mathsf{R}_{\mathsf{lin}}$. To verify these transformations in DimSum, one can use additional wrappers to make sure the events do match up. For example, in the case of *Mem2Reg*, we *cannot prove* $[\![\mathsf{R}_{\mathsf{opt}}]\!]_{\mathsf{r}} \preceq [\![\mathsf{R}_{\mathsf{lin}}]\!]_{\mathsf{r}}$ because of the event mismatch, but we *can prove* $[\![\mathsf{R}_{\mathsf{opt}}]\!]_{\mathsf{r}} \preceq \lceil [\![\mathsf{R}_{\mathsf{lin}}]\!]_{\mathsf{r}} \rceil_{\mathsf{r} \rightleftharpoons \mathsf{r}}$ where the wrapper $\lceil \cdot \rceil_{\mathsf{r} \rightleftharpoons \mathsf{r}}$ transforms the events of $[\![\mathsf{R}_{\mathsf{lin}}]\!]_{\mathsf{r}}$ such that they match up with $[\![\mathsf{R}_{\mathsf{opt}}]\!]_{\mathsf{r}}$. Concretely, the $\lceil \cdot \rceil_{\mathsf{r} \rightleftharpoons \mathsf{r}}$ wrapper ensures that *private* memory locations (*i.e.*, local variables that have never been shared with the environment via function arguments or return values) are not part of the events, and thus *Mem2Reg* is allowed to optimize them away.

While wrappers such as $\lceil \cdot \rceil_{\mathsf{r} \rightleftharpoons \mathsf{r}}$ enable the verification of more program transformations, we also have to make sure that they do not allow *too many* transformations (*i.e.*, incorrect transformations). This constraint is handled implicitly by the compiler correctness statement COMPILER-CORRECT—it

[16] To simplify the presentation, the rule COMPILER-CORRECT omits some technical details relevant for the translation between Rec-function names and the corresponding **Asm**-instruction addresses.

does not mention any wrappers other than $\lceil\cdot\rceil_{\mathsf{r}\rightleftharpoons\mathsf{a}}$, and in order to use a new wrapper such as $\lceil\cdot\rceil_{\mathsf{r}\rightleftharpoons\mathsf{r}}$, we have to show that the additional transformations it enables are also allowed by the wrapper $\lceil\cdot\rceil_{\mathsf{r}\rightleftharpoons\mathsf{a}}$. We call this property "*vertical compositionality*" of the two wrappers (REC-TO-ASM-VERTICAL below) and it allows us to prove $\lceil\lceil[\![\mathsf{R}_{\mathsf{lin}}]\!]_{\mathsf{r}}\rceil_{\mathsf{r}\rightleftharpoons\mathsf{r}}\rceil_{\mathsf{r}\rightleftharpoons\mathsf{a}} \preceq \lceil[\![\mathsf{R}_{\mathsf{lin}}]\!]_{\mathsf{r}}\rceil_{\mathsf{r}\rightleftharpoons\mathsf{a}}$ in the refinement chain of COMPILER-CORRECT.

Vertical compositionality. Vertical compositionality in DimSum means not only proving transitivity of the simulation relation ($\preceq$), but also that certain "intermediate wrappers" can be eliminated. For instance, the vertical compositionality result of $\lceil\cdot\rceil_{\mathsf{r}\rightleftharpoons\mathsf{r}}$ in this compiler correctness proof is given by:

$$\text{REC-TO-ASM-VERTICAL}\ \lceil\lceil\mathsf{M}\rceil_{\mathsf{r}\rightleftharpoons\mathsf{r}}\rceil_{\mathsf{r}\rightleftharpoons\mathsf{a}} \preceq \lceil\mathsf{M}\rceil_{\mathsf{r}\rightleftharpoons\mathsf{a}}$$

This theorem, like most vertical compositionality theorems, is difficult to prove, since we need to show that certain Rec-level memory transformations do not change the behavior of M from the perspective of **Asm** in any meaningful way.

Note that in other approaches to multi-language semantics, vertical compositionality typically either requires composing simulation relations [Neis et al. 2015; Stewart et al. 2015] or proving the transitivity of contextual refinement [Perconti and Ahmed 2014]. In contrast, in DimSum, the (Kripke) wrappers $\lceil M\rceil_X$ effectively assume the role of "simulation conventions" or "simulation invariants" in a (Kripke) simulation relation (*e.g.*, relations on memories and values), but they do so as *a module combinator*. As a result, proving vertical compositionality in DimSum is not necessarily simpler, but it is more localized: if we want vertical compositionality of two transformations (*i.e.*, two wrappers), then we prove a single simulation (*e.g.*, REC-TO-ASM-VERTICAL), and no other parts of the framework are affected, since they are compatible with simulation.

The $\lceil\cdot\rceil_{\mathsf{r}\rightleftharpoons\mathsf{r}}$ wrapper. Let us now turn to the definition of the wrapper $\lceil\cdot\rceil_{\mathsf{r}\rightleftharpoons\mathsf{r}}$, which is an instance of the generic combinator $\lceil M\rceil_X$ (from §3.3). As explained above, the purpose of $\lceil\cdot\rceil_{\mathsf{r}\rightleftharpoons\mathsf{r}}$ is to enable optimizing memory locations that are kept private and never shared with the environment (*e.g.*, via function arguments or return values). To this end, we instantiate $\lceil M\rceil_X$ with a separation logic $\mathcal{L}$ inspired by Gäher et al. [2022] that supports assertions for both persistent ownership of shared locations ($\ell_1 \leftrightarrow \ell_2$) and exclusive ownership of private memory locations ($\ell_1 \mapsto_{\mathsf{E}} \mathsf{v}$ and $\ell_2 \mapsto_{\mathsf{I}} \mathsf{v}$). The assertion $\ell_1 \leftrightarrow \ell_2$ states that location ℓ_1 in the *external memory* (*i.e.*, the memory of $\lceil\mathsf{M}\rceil_{\mathsf{r}\rightleftharpoons\mathsf{r}}$) always corresponds to ℓ_2 in the *internal memory* (*i.e.*, the memory of M). The assertion $\ell \mapsto_{\mathsf{E}} \mathsf{v}$ conveys exclusive ownership of the locations ℓ in the external memory, and $\ell \mapsto_{\mathsf{I}} \mathsf{v}$ of ℓ in the internal memory. Additionally, the separation logic provides the assertion $\mathsf{inv}(\mathsf{m}_1, \mathsf{m}_2)$ which connects "$\ell_1 \leftrightarrow \ell_2$", "$\ell \mapsto_{\mathsf{E}} \mathsf{v}$", and "$\ell \mapsto_{\mathsf{I}} \mathsf{v}$" to the heaps m_1 and m_2 such that the heaps overlap in the way described by the assertions. We then instantiate the relations $\mathsf{e}_{\mathsf{I}} \rightharpoonup \mathsf{e}_{\mathsf{E}}$ and $\mathsf{e}_{\mathsf{I}} \leftharpoonup \mathsf{e}_{\mathsf{E}}$ with:

$$\mathsf{e}_{\mathsf{I}} \rightharpoonup \mathsf{e}_{\mathsf{E}}, \mathsf{e}_{\mathsf{I}} \leftharpoonup \mathsf{e}_{\mathsf{E}} \triangleq \mathsf{type}(\mathsf{e}_{\mathsf{I}}) = \mathsf{type}(\mathsf{e}_{\mathsf{E}}) * \mathsf{inv}(\mathsf{mem}(\mathsf{e}_{\mathsf{I}}), \mathsf{mem}(\mathsf{e}_{\mathsf{E}})) * \forall(\mathsf{v}_1, \mathsf{v}_2) \in \mathsf{vals}(\mathsf{e}_{\mathsf{I}}, \mathsf{e}_{\mathsf{E}}).\ \mathsf{v}_1 \leftrightarrow \mathsf{v}_2$$

This definition consists of three parts: First, the type of the events (*i.e.*, call or return) has to match. Second, the invariant inv has to hold for the memories contained in the events. Third, the values of events (*e.g.*, function arguments and return values) must be related (*i.e.*, $\ell_1 \leftrightarrow \ell_2$ for locations and equality for integers and Booleans).

Let us now consider how the wrapper $\lceil\cdot\rceil_{\mathsf{r}\rightleftharpoons\mathsf{r}}$ enables the verification of the *Mem2Reg* pass, which transforms $\mathsf{R}_{\mathsf{lin}}$ to $\mathsf{R}_{\mathsf{opt}}$. Recall that this transformation replaces a local variable allocated at some location ℓ in $\mathsf{R}_{\mathsf{lin}}$ with a let-binding without corresponding allocation in $\mathsf{R}_{\mathsf{opt}}$. To justify this transformation, we need to prove that the value stored at ℓ in $\mathsf{R}_{\mathsf{lin}}$ always corresponds to the let-bound value in $\mathsf{R}_{\mathsf{opt}}$ and, in particular, remains constant across external function calls. To this end, we use the $\ell \mapsto_{\mathsf{I}} \mathsf{v}$ assertion, which we obtain at the start of the function when ℓ is allocated on

the heap, and we keep it in the privately owned part P_1 of the combinator $\lceil M \rceil_X$ (see §3.3). That is, $\ell \mapsto_{\mathsf{I}} \mathsf{v}$ allows us to track the precise value of ℓ and ensure that it corresponds to the let-bound value in the optimized program. When calling the environment, we do not need to give up $\ell \mapsto_{\mathsf{I}} \mathsf{v}$, because the location is never exposed and thus never appears in $\mathsf{e}_{\mathsf{I}} \rightharpoonup \mathsf{e}_{\mathsf{E}}$ (otherwise the optimization does not fire). Instead, we can keep $\ell \mapsto_{\mathsf{I}} \mathsf{v}$ in the privately owned part of the module P_1 and the definition of wrap ensures that $\ell \mapsto_{\mathsf{I}} \mathsf{v}$ holds across the function call. Thus, we know ℓ still points to v after the function call, which allows us to complete the verification of the *Mem2Reg* pass.

6 RELATED WORK

CompCert-based approaches to multi-language verification. Although CompCert's original correctness statement [Leroy 2006] only concerns whole programs, it inspired a long line of work on multi-language verification [Beringer et al. 2014; Ramananandro et al. 2015; Stewart et al. 2015; Kang et al. 2016; Gu et al. 2015; Wang et al. 2019; Song et al. 2020; Koenig and Shao 2021]. The approach most closely related to DimSum is CompCertO [Koenig and Shao 2021], since its game semantics-based approach has parallels to the event-based approach of DimSum—*e.g.*, CompCertO's language interfaces play a similar role to the event types of DimSum. To scale to the full extent of CompCert, CompCertO makes design choices specific to the languages of CompCert. In particular, it provides only a single linking operator that enforces a well-bracketed call structure (unlike $\mathsf{M}_1 \oplus_{\mathsf{coro}} \mathsf{M}_2$), studies only transition systems without private state across function invocations (unlike $\lceil \cdot \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}$), and—even though CompCertO's underlying definition of simulation convention is independent of the memory model—considers only languages with the same memory model (unlike Rec and **Asm**).

Compositional CompCert [Stewart et al. 2015], Ramananandro et al. [2015], and CompCertM [Song et al. 2020] achieve multi-language linking by imposing a common interaction protocol between all languages. This works well in their setting since all CompCert languages share a notion of values and memory, but it is unclear how to scale this setup to more heterogeneous languages like Rec and **Asm**. Similar to DimSum, Ramananandro et al. use events that contain the complete program memory. They define linking on traces of call events (*i.e.*, "behaviors") and prove equivalence between syntactic and semantic linking similar to ASM-LINK-SYN and REC-LINK-SYN. However, their traces erase the branching structure of the program. In DimSum, sensitivity to branching is crucial due to the presence of both demonic and angelic non-determinism (see §3.2), so we define linking directly on transition systems (*i.e.*, "modules").

CompCertX and (Concurrent) Certified Abstraction Layers [Gu et al. 2015; Wang et al. 2019; Gu et al. 2018; Vale et al. 2022] have been successfully deployed in the verification of the CertiKOS verified operating system. However, they impose restrictions on the interaction between different components, such as forbidding mutual recursion and certain memory sharing patterns. In contrast, our semantic linking operators $\mathbf{M_1} \oplus_{\mathsf{a}} \mathbf{M_2}$ and $\mathsf{M}_1 \oplus_{\mathsf{r}} \mathsf{M}_2$ allow mutual recursion and memory sharing.

Syntactic multi-languages. Syntactic multi-languages [Matthews and Findler 2007; Perconti and Ahmed 2014; Patterson et al. 2017; Mates et al. 2019; Patterson et al. 2022] embed source, target, and intermediate languages into a common multi-language (with "boundary" terms to translate between languages) and then use the multi-language to, among other things, state and verify compiler correctness theorems. As this line of work demonstrates, syntactic multi-languages scale well to typed, higher-order languages. In this paper, we have put the focus on different kinds of languages: untyped, low-level languages comparable to C and assembly.

Specifications in syntactic multi-languages use contextual equivalence, which is canonical but has the downside of including the operations (and semantics) of all involved languages in every

specification. In contrast, when we prove $[\![\mathsf{main} \cup_{\mathsf{r}} \mathsf{memmove}]\!]_{\mathsf{r}} \oplus_{\mathsf{r}} [\![\mathsf{locle}_{\mathsf{spec}}]\!]_{\mathsf{s}} \preceq [\![\mathsf{main}_{\mathsf{spec}}]\!]_{\mathsf{s}}$ in §2, we only care about the semantics of Rec, not that the modules are embedded in an **Asm** context.

Mates et al. [2019] prove correctness of a compiler from a source language without call/cc to a target language with call/cc and show that compiled code can be linked with a thread library loosely similar to our coroutine library (§4.3). They do not provide a high-level abstraction like $M_1 \oplus_{\mathsf{coro}} M_2$ and instead require clients to reason about the implementation of the library. In their case, the distinction does not matter much because the target language is reasonably high-level, but for Rec and **Asm**, it would involve reasoning about low-level stack and register manipulation.

Recently, Patterson et al. [2022] proved type safety for several, very different multi-languages by giving a realizability model in an untyped target language. While effective for type safety, the downside of this approach is that most reasoning happens at the target language. In contrast, an important goal of DimSum is to lift reasoning to source-level languages as shown by §2 and §4.3.

Pilsner. Building on the work of Hur and collaborators [Benton and Hur 2009, 2010; Hur and Dreyer 2011; Hur et al. 2012], Pilsner [Neis et al. 2015] verifies two compilers from a higher-order stateful source language to an assembly target language and shows that the compiled programs can be safely linked. However, Pilsner prohibits linking with target-level libraries whose observable functionality is inexpressible in the source; as such, it rules out the examples in §2 and §4.3.

Other approaches. The Cito compiler [Wang et al. 2014; Pit-Claudel et al. 2020] simplifies its compositional compiler correctness statement by requiring the user to provide specifications for all external functions. In contrast, while DimSum supports giving specifications for low-level libraries as seen in §2, they are not required by our compiler correctness theorem.

Conditional Contextual Refinement (CCR) [Song et al. 2023] uses dual (demonic and angelic) nondeterminism to encode a wrapper that can transform the values of function arguments and results and enforce separation logic preconditions and postconditions. DimSum's wrappers are inspired by this idea but apply it to interoperation between different languages and memory models instead of program verification. While CCR allows linking between different languages (*e.g.*, between an implementation language and a specification language), this linking shares the drawbacks of some CompCert-based approaches in that it is restricted to languages with well-bracketed call structure.

Properties of wrappers. The idea of translating between different languages via wrappers originates in the work on multi-language semantics [Matthews and Findler 2007; Ahmed and Blume 2011]. This prior work on syntactic wrappers identified desirable properties for such wrappers, including boundary cancellation [Perconti and Ahmed 2014] and embedding-projection pairs [New and Ahmed 2018]. It would be interesting to investigate how these properties can be phrased in terms of DimSum's semantic wrappers.

Process algebra. DimSum's way of modeling and relating the semantics of modular components takes inspiration from the π-calculus [Milner et al. 1992] and Communicating Sequential Processes (CSP) [Hoare 1978]. The π-calculus and its predecessor CCS pioneered the idea of characterizing the behavior of a component in an arbitrary context using labeled transition systems, where the labels represented potential interaction with the environment, and comparing behavior using (bi-)simulations. Notably, CSP and Session-Typed variants of π-calculus [Padovani 2010] include dual internal and external choice constructs. They are, however, modeling concurrent process interaction (*i.e.*, offering and selecting among a set of actions), and not, as in DimSum, rely/guarantee-style contracts with the environment.

Fully abstract traces. The work on fully abstract trace semantics [Jeffrey and Rathke 2005; Laird 2007; Abadi and Plotkin 2010; Patrignani et al. 2015] uses events to describe the interaction between program components similar to the events of **Asm** and Rec. However, prior work either focuses on proving full abstraction of the trace semantics or uses fully abstract trace semantics to prove full abstraction of a compiler, not for reasoning about multi-language programs.

7 FUTURE WORK

This paper presents a first step towards exploring a decentralized approach for multi-language verification and exercises it on the Rec and **Asm** languages. In future work, it would be interesting to explore how to scale DimSum's approach to bigger languages with different features. Some language features might be representable by the existing combinators presented in §3.3, but others might require defining new kinds of combinators—*e.g.*, concurrency, since the existing combinators only provide sequential interleaving. However, the core concept of DimSum—modules as transition systems which communicate via events—is in principle quite general (as shown *e.g.*, by the field of process algebra), so we are hopeful that DimSum will provide a viable foundation for the verification of realistic multi-language programs.

ACKNOWLEDGMENTS

We would like to thank Ralf Jung and Chung-Kil Hur for many insightful discussions, and the anonymous reviewers for their helpful feedback. This work was funded in part by the Dutch Research Council (NWO), project 016.Veni.192.259, by Google PhD Fellowships for the first and second authors, by awards from Android Security's ASPIRE program and from Google Research, and by a European Research Council (ERC) Consolidator Grant for the project "PERSIST" under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 101003349).

DATA AVAILABILITY STATEMENT

The Coq development and appendix for this paper can be found in Sammler et al. [2023b]. The current development version of DimSum is linked from the project webpage at https://plv.mpi-sws.org/dimsum/.

REFERENCES

Martín Abadi and Gordon D. Plotkin. 2010. On Protection by Layout Randomization. In *CSF*. IEEE Computer Society, 337–351. https://doi.org/10.1109/CSF.2010.30

Amal Ahmed and Matthias Blume. 2011. An equivalence-preserving CPS translation via multi-language semantics. In *ICFP*. ACM, 431–444. https://doi.org/10.1145/2034773.2034830

Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Y. Vardi. 1998. Alternating Refinement Relations. In *CONCUR (LNCS, Vol. 1466)*. Springer, 163–178. https://doi.org/10.1007/BFb0055622

Nick Benton and Chung-Kil Hur. 2009. Biorthogonality, step-indexing and compiler correctness. In *ICFP*. ACM, 97–108. https://doi.org/10.1145/1596550.1596567

Nick Benton and Chung-Kil Hur. 2010. *Realizability and compositional compiler correctness for a polymorphic language*. Technical Report MSR-TR-2010-62. Microsoft Research. https://sf.snu.ac.kr/publications/cccmsrtr.pdf

Lennart Beringer, Gordon Stewart, Robert Dockins, and Andrew W. Appel. 2014. Verified Compilation for Shared-Memory C. In *ESOP (LNCS, Vol. 8410)*. Springer, 107–127. https://doi.org/10.1007/978-3-642-54833-8_7

Ashok K. Chandra, Dexter Kozen, and Larry J. Stockmeyer. 1981. Alternation. *J. ACM* 28, 1 (1981), 114–133. https://doi.org/10.1145/322234.322243

Robert W. Floyd. 1967. Nondeterministic Algorithms. *J. ACM* 14, 4 (1967), 636–644. https://doi.org/10.1145/321420.321422

Carsten Fritz and Thomas Wilke. 2005. Simulation relations for alternating Büchi automata. *Theor. Comput. Sci.* 338, 1-3 (2005), 275–314. https://doi.org/10.1016/j.tcs.2005.01.016

Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. 2022. Simuliris: a separation logic framework for verifying concurrent program optimizations. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–31. https://doi.org/10.1145/3498689

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *POPL*. ACM, 595–608. https://doi.org/10.1145/2676726.2676975

Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified concurrent abstraction layers. In *PLDI*. ACM, 646–661. https://doi.org/10.1145/3192366.3192381

C. A. R. Hoare. 1978. Communicating Sequential Processes. *Commun. ACM* 21, 8 (1978), 666–677. https://doi.org/10.1145/359576.359585

Chung-Kil Hur and Derek Dreyer. 2011. A Kripke logical relation between ML and assembly. In *POPL*. ACM, 133–146. https://doi.org/10.1145/1926385.1926402

Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. 2012. The marriage of bisimulations and Kripke logical relations. In *POPL*. ACM, 59–72. https://doi.org/10.1145/2103656.2103666

Alan Jeffrey and Julian Rathke. 2005. Java Jr: Fully Abstract Trace Semantics for a Core Java Language. In *ESOP (Lecture Notes in Computer Science, Vol. 3444)*. Springer, 423–438. https://doi.org/10.1007/978-3-540-31987-0_29

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. ACM, 637–650. https://doi.org/10.1145/2676726.2676980

Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. 2016. Lightweight verification of separate compilation. In *POPL*. ACM, 178–190. https://doi.org/10.1145/2837614.2837642

Jérémie Koenig and Zhong Shao. 2021. CompCertO: compiling certified open C components. In *PLDI*. ACM, 1095–1109. https://doi.org/10.1145/3453483.3454097

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A general, extensible modal framework for interactive proofs in separation logic. *Proc. ACM Program. Lang.* 2, ICFP (2018), 77:1–77:30. https://doi.org/10.1145/3236772

Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017a. The Essence of Higher-Order Concurrent Separation Logic. In *ESOP (LNCS, Vol. 10201)*. Springer, 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017b. Interactive proofs in higher-order concurrent separation logic. In *POPL*. ACM, 205–217. https://doi.org/10.1145/3009837.3009855

James Laird. 2007. A Fully Abstract Trace Semantics for General References. In *ICALP (LNCS, Vol. 4596)*. Springer, 667–679. https://doi.org/10.1007/978-3-540-73420-8_58

Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*. ACM, 42–54. https://doi.org/10.1145/1111037.1111042

Xavier Leroy and Sandrine Blazy. 2008. Formal verification of a C-like memory model and its uses for verifying program transformations. *JAR* 41, 1 (2008), 1–31. https://doi.org/10.1007/s10817-008-9099-0

Phillip Mates, Jamie Perconti, and Amal Ahmed. 2019. Under Control: Compositionally Correct Closure Conversion with Mutable State. In *PPDP*. ACM, 16:1–16:15. https://doi.org/10.1145/3354166.3354181

Jacob Matthews and Robert Bruce Findler. 2007. Operational semantics for multi-language programs. In *POPL*. ACM, 3–10. https://doi.org/10.1145/1190216.1190220

Robin Milner. 1999. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press.

Robin Milner, Joachim Parrow, and David Walker. 1992. A Calculus of Mobile Processes, I/II. *Inf. Comput.* 100, 1 (1992), 1–40. https://doi.org/10.1016/0890-5401(92)90008-4

Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. 2015. Pilsner: a compositionally verified compiler for a higher-order imperative language. In *ICFP*. ACM, 166–178. https://doi.org/10.1145/2784731.2784764

Max S. New and Amal Ahmed. 2018. Graduality from embedding-projection pairs. *Proc. ACM Program. Lang.* 2, ICFP (2018), 73:1–73:30. https://doi.org/10.1145/3236768

Luca Padovani. 2010. Session Types = Intersection Types + Union Types. In *ITRS (EPTCS, Vol. 45)*. 71–89. https://doi.org/10.4204/EPTCS.45.6

Marco Patrignani. 2020. Why Should Anyone use Colours? or, Syntax Highlighting Beyond Code Snippets. *CoRR* abs/2001.11334 (2020). https://arxiv.org/abs/2001.11334

Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. 2015. Secure Compilation to Protected Module Architectures. *ACM Trans. Program. Lang. Syst.* 37, 2 (2015), 6:1–6:50. https://doi.org/10.1145/2699503

Daniel Patterson and Amal Ahmed. 2019. The next 700 compiler correctness theorems (functional pearl). *Proc. ACM Program. Lang.* 3, ICFP (2019), 85:1–85:29. https://doi.org/10.1145/3341689

Daniel Patterson, Noble Mushtak, Andrew Wagner, and Amal Ahmed. 2022. Semantic soundness for language interoperability. In *PLDI*. ACM, 609–624. https://doi.org/10.1145/3519939.3523703

Daniel Patterson, Jamie Perconti, Christos Dimoulas, and Amal Ahmed. 2017. FunTAL: reasonably mixing a functional language with assembly. In *PLDI*. ACM, 495–509. https://doi.org/10.1145/3062341.3062347

James T. Perconti and Amal Ahmed. 2014. Verifying an Open Compiler Using Multi-language Semantics. In *ESOP (LNCS, Vol. 8410)*. Springer, 128–148. https://doi.org/10.1007/978-3-642-54833-8_8

Clément Pit-Claudel, Peng Wang, Benjamin Delaware, Jason Gross, and Adam Chlipala. 2020. Extensible Extraction of Efficient Imperative Programs with Foreign Functions, Manually Managed Memory, and Proofs. In *IJCAR (LNCS, Vol. 12167)*. 119–137. https://doi.org/10.1007/978-3-030-51054-1_7

Tahina Ramananandro, Zhong Shao, Shu-Chun Weng, Jérémie Koenig, and Yuchen Fu. 2015. A Compositional Semantics for Verified Separate Compilation and Linking. In *CPP*. ACM, 3–14. https://doi.org/10.1145/2676724.2693167

Ingrid Rewitzky. 2003. Binary Multirelations. In *Theory and Applications of Relational Structures as Knowledge Instruments*. LNCS, Vol. 2929. Springer, 256–271. https://doi.org/10.1007/978-3-540-24615-2_12

A. W. Roscoe. 2010. *Understanding Concurrent Systems*. Springer. https://doi.org/10.1007/978-1-84882-258-0

Michael Sammler, Angus Hammond, Rodolphe Lepigre, Brian Campbell, Jean Pichon-Pharabod, Derek Dreyer, Deepak Garg, and Peter Sewell. 2022. Islaris: verification of machine code against authoritative ISA semantics. In *PLDI*. ACM, 825–840. https://doi.org/10.1145/3519939.3523434

Michael Sammler, Simon Spies, Youngju Song, Emanuele D'Osualdo, Robbert Krebbers, Deepak Garg, and Derek Dreyer. 2023a. DimSum: A Decentralized Approach to Multi-language Semantics and Verification (Appendix). https://doi.org/10.5281/zenodo.7306313 Project webpage: https://plv.mpi-sws.org/dimsum/.

Michael Sammler, Simon Spies, Youngju Song, Emanuele D'Osualdo, Robbert Krebbers, Deepak Garg, and Derek Dreyer. 2023b. DimSum: A Decentralized Approach to Multi-language Semantics and Verification (Coq development). https://doi.org/10.5281/zenodo.7306313 Project webpage: https://plv.mpi-sws.org/dimsum/.

Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. 2020. CompCertM: CompCert with C-assembly linking and lightweight modular verification. *Proc. ACM Program. Lang.* 4, POPL (2020), 23:1–23:31. https://doi.org/10.1145/3371091

Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer. 2023. Conditional Contextual Refinement. In *POPL*. ACM. https://doi.org/10.1145/3571232

Christoph Sprenger, Tobias Klenze, Marco Eilers, Felix A. Wolf, Peter Müller, Martin Clochard, and David A. Basin. 2020. Igloo: soundly linking compositional refinement and separation logic for distributed system verification. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 152:1–152:31. https://doi.org/10.1145/3428220

Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *POPL*. ACM, 275–287. https://doi.org/10.1145/2676726.2676985

Andrew S. Tanenbaum and Herbert Bos. 2014. *Modern Operating Systems* (4th ed.). Prentice Hall Press, USA. https://dl.acm.org/doi/book/10.5555/2655363

Arthur Oliveira Vale, Paul-André Melliès, Zhong Shao, Jérémie Koenig, and Léo Stefanesco. 2022. Layered and object-based game semantics. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–32. https://doi.org/10.1145/3498703

Moshe Y. Vardi. 1995. Alternating Automata and Program Verification. In *Computer Science Today*. LNCS, Vol. 1000. Springer, 471–485. https://doi.org/10.1007/BFb0015261

Peng Wang, Santiago Cuellar, and Adam Chlipala. 2014. Compiler verification meets cross-language linking via data abstraction. In *OOPSLA*. ACM, 675–690. https://doi.org/10.1145/2660193.2660201

Yuting Wang, Pierre Wilke, and Zhong Shao. 2019. An abstract stack based approach to verified compositional compilation to machine code. *Proc. ACM Program. Lang.* 3, POPL (2019), 62:1–62:30. https://doi.org/10.1145/3290375

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 51:1–51:32. https://doi.org/10.1145/3371119

Received 2022-07-07; accepted 2022-11-07