Intrinsically Typed Compilation with Nameless Labels

ARJEN ROUVOET, Delft University of Technology, The Netherlands

ROBBERT KREBBERS, Radboud University and Delft University of Technology, The Netherlands

EELCO VISSER, Delft University of Technology, The Netherlands

To avoid compilation errors it is desirable to verify that a compiler is *type correct*—i.e., given well-typed source code, it always outputs well-typed target code. This can be done *intrinsically* by implementing it as a function in a dependently typed programming language, such as Agda. This function manipulates data types of well-typed source and target programs, and is therefore type correct by construction. A key challenge in implementing an intrinsically typed compiler is the representation of labels in bytecode. Because label names are global, bytecode typing appears to be inherently a non-compositional, whole-program property. The individual operations of the compiler do not preserve this property, which requires the programmer to reason about labels, which spoils the compiler definition with proof terms.

In this paper, we address this problem using a new *nameless* and *co-contextual* representation of typed global label binding, which *is* compositional. Our key idea is to use *linearity* to ensure that all labels are defined exactly once. To write concise compilers that manipulate programs in our representation, we develop a linear, dependently typed, shallowly embedded language in Agda, based on separation logic. We show that this language enables the concise specification and implementation of intrinsically typed operations on bytecode, culminating in an intrinsically typed compiler for a language with structured control-flow.

1 INTRODUCTION

Compilers that go wrong turn correct source programs into incorrect target programs. Verifying *functional* correctness of compilers offers a complete solution, proving a strong relation between the semantics of the source and the target of compilation. The most extensive and well-known projects in this direction are CompCert [Leroy 2009] and CakeML [Kumar et al. 2014], which provide a fully verified compiler for the C and ML programming language, respectively. The great confidence in such compilers comes at the price of the research and development that is required to establish its correctness. Projects like CompCert and CakeML are the result of a decade of work into specifying the semantics of the (intermediate) languages involved in the compiler, and specifying and proving the simulations between these semantics. If we want to avoid these costs of functional

Authors' addresses: Arjen Rouvoet, Delft University of Technology, The Netherlands, a.j.rouvoet@tudelft.nl; Robbert Krebbers, Radboud University and Delft University of Technology, The Netherlands, mail@robbertkrebbers.nl; Eelco Visser, Delft University of Technology, The Netherlands, e.visser@tudelft.nl.

```
Compiler a  =  Label  →  Label × Bytecode × a
compile  :  Exp  →  Compiler ()
compile (if c then e₁ else e₂)  =  do
    compile c
    ℓthen  ←  freshLabel
    tell [ iftrue ℓthen ]
    compile e₂
    ℓend  ←  freshLabel
    tell [ goto ℓend ]
    attach ℓthen (compile e₁)
    attach ℓend  (return ())
```

Pre stack type	Label	Instruction(s)	Post stack type
ψ		$[\![c]\!]$	boolean :: ψ
boolean :: ψ		iftrue ℓ_{then}	ψ
ψ		$[\![e_2]\!]$	$a :: \psi$
$a :: \psi$		goto ℓ_{end}	$a :: \psi$
ψ	ℓ_{then}	$[\![e_1]\!]$	$a :: \psi$
$a :: \psi$	ℓ_{end}	nop	$a :: \psi$

Fig. 1. The if c then e_1 else e_2 case of a monadic compiler from an expression language to bytecode, and a table with the corresponding compilation template. The table assumes c : boolean, e_1 : a, and e_2 : a, and uses $[\![e]\!]$ to denote the instructions obtained by compilation of e.

verification, but still want to avoid a large class of miscompilations, we can instead (mechanically) verify *type correctness* of a compiler [Abel 2020; Bélanger et al. 2015; Chlipala 2007; Guillemette and Monnier 2008]. This is a weaker property than functional correctness, but is much easier to specify and prove, as it does not involve the dynamic semantics of the source and target languages of transformations. At the same time, it still rules out many bugs in a compiler, because we are guaranteed that well-typed target code does not "go wrong" [Milner 1978].

One method to verify type correctness of a compiler is by implementing it in a proof assistant and proving the type correctness property *extrinsically*—i.e., by writing a separate proof that reasons about the implementation of the compiler. The separation of the implementation and the type correctness proof, however, has a disadvantage: type errors and verification issues are not discovered interactively during the development of the compiler implementation. Both these concerns can be addressed by integrating the type correctness proof in the compiler definition. This can be accomplished by defining datatypes of *well-typed* source and target languages in a dependently typed language, like Agda [Norell 2009], and implementing the compiler as a function from the former to the latter. We call this an *intrinsically typed* compiler [McKinna and Wright 2006; Reynolds 2000], because it verifies the type correctness property of the compiler internally, as part of the definition.

The benefits of intrinsically typed programming are weighed against the cost of integrating the type correctness proof in the program. The costs consist of the burden that the proof imposes on the programmer and on the readability of the program. Previous work on intrinsically typed interpreters (for example [Augustsson and Carlsson 1999; Benton et al. 2012; Poulsen et al. 2018; Rouvoet et al. 2020a]) has shown that intrinsically typed *interpreters* can sometimes achieve a favorable cost-benefit balance. In this paper we extend the state of the art to the definition of intrinsically typed *compilers* that target a low-level representation of bytecode with labels.

Compiler type correctness. Consider the compilation of the expression if c then s_1 else s_2 in Fig. 1. It is written in a declarative style that exactly mirrors the compilation template shown on the right. The declarative nature of the untyped compiler in Fig. 1 is accomplished by writing it as a computation in the monad Compiler, which combines a writer monad with state. The writer part collects the generated instructions, and the state part provides a supply of fresh labels. The (writer-) monad operation tell is used to output a given sequence of instructions, and the operation attach is

used to label the first instruction of the output of a compiler computation. Recursive invocations of the compiler are used to produce the output for the sub-expressions.

Assuming that the input expression is well typed, we now ask the question: is the result of the compiler well typed? The typing of instructions is based on their effect on the state of the machine that executes them. For the set of instructions shown, the relevant part of the state is the operand stack. Type correctness of the bytecode produced by the compiler requires the typing of the stack to match between instructions and subsequent instructions. For most instructions this is the next instruction in the sequence, but for (conditional) jumps (i.e., **iftrue** and **goto**) this is an instruction marked by a label. These labels must be *well bound*—i.e., they must unambiguously reference one instruction in the output. Concretely this means that labels must be declared *exactly once*.

To see that the compilation of conditional expressions in Fig. 1 is indeed type correct, we need to make use of the *stack invariant* of expression compilation, which states that executing the bytecode for an expression e of type a in an initial stack configuration typed ψ will leave a single value of type a behind on top of the initial stack ψ. Using this invariant we can construct the typing for the bytecode template shown in the table in Fig. 1, and verify that it satisfies the type correctness criterion. For example, **iftrue** ℓ_{then} is followed either by the subsequent instructions for the "else" branch, or the bytecode labeled ℓ_{then} for the "then" branch. Both expect a stack typed ψ, matching the type of the stack after popping the condition.

The problem with intrinsically typed compilation. Unfortunately, even if we implement this compiler in a strongly typed language, like Haskell, the programmer can still make mistakes that are essentially type errors. For example, the type checker will not enforce that we only pop from non-empty stacks. Nor does it guard against jump instructions that references labels that are never attached to an instruction. If the compiler outputs such a jump, then the resulting bytecode program is not well bound and thus also not well typed. We want to rule out all such errors and guarantee that any compilation function that is type correct in the host language outputs well-typed bytecode.

To rule out type errors, we can try to make use of an intrinsically typed representation of bytecode. Using dependent types, we can strengthen the types of the bytecode representation in the host language so that they express the typing rules of the embedded bytecode language. Similarly, we can strengthen the types of the compiler operations so that they express the type invariants of the code transformations. The goal of this paper is to accomplish exactly that. Importantly, we want to do this in such a way that we can use these new and improved operations to define compile without spoiling its declarative nature. Key to this is ensuring that the formulation of bytecode typing is *compositional*, so that the well-typed compiler operations can be composed into a well-typed compiler with little manual proof effort. In doing this, we encounter two problems:

- Many operations of the compiler are simply not well bound in isolation. For example, the third operation tell [**iftrue** ℓ_{then}] in the compiler in Fig. 1, which outputs the **iftrue** instruction, produces a jump with a dangling label reference ℓ_{then}. That is, it is type correct only provided that the label ℓ_{then} has previously been attached or will eventually be attached to some bytecode in the output.
- Even though the compilation in Fig. 1 appears locally well bound, type correctness crucially depends on the generated labels ℓ_{then} and ℓ_{end} being *fresh* for the *entire* compilation. That is, type correctness depends not only on which *concrete* labels we fill in for the symbolic labels in the template, but also on whether those are different from the labels used in the surrounding bytecode, and the bytecode of the condition and branch expressions.

In other words, bytecode typing appears to be a whole-program property, that is simply not preserved by the individual operations of our compiler. This obstructs a compositional formulation and makes it difficult to define the intrinsically typed operations of compilation. In §2, we show

that such a non-compositional typing result in intrinsically typed compilers that are bloated with proofs of side conditions. This puts a burden on the programmer to prove these side conditions, and spoils the declarative nature of the compiler.

Key idea 1: Nameless co-contextual bytecode. If bytecode typing—and global binding in particular— is indeed inherently anti-compositional, how can we expect to obtain a type correct compiler by mere composition of the individual operations? We identify two reasons that make bytecode and its typing anti-compositional: (1) the *contextual* formulation of traditional typing rules, and (2) the use of names to represent global label binding in bytecode terms. To address these issues, we propose a new *nameless* and *co-contextual* [Erdweg et al. 2015] typing of bytecode.

In traditional typing rules, one deals with bound names (like labels) using contexts. Contexts are inherently non-local, as they summarize the *surroundings* of a term. In bytecode, the scope of labels is not restricted and consequently the label context contains *all* labels in the program. This forces us into taking a whole-program perspective. For example, concatenating two independent, contextually typed bytecode fragments requires *extrinsic* evidence that the sets of labels that are defined in these fragments are disjoint. This requires non-local reasoning—e.g., using a supply of provably, globally fresh label names. It also requires weakening of the fragments because we bring more labels into scope using the concatenation.

Instead, we will use *co-contexts* to deal with bound names, which only describe the types of the labels that are *exports* (i.e., label binders) and *imports* (i.e., the label references/jumps) of a bytecode fragment. Co-contexts are *principal* [Jim 1996], in the sense that they are the smallest sets of exports and imports that work. This means that weakening is never required. By additionally using a *nameless* representation, there can also not be accidental overlap between exports. This means that we can *always* concatenate bytecode fragments without proving side conditions. The co-context of the composition is simply the union of the exports and imports. Our representation is inspired by the nameless co-de-Bruijn representation of lexical binding by McBride [2018], and the nameless representation of linear references by Rouvoet et al. [2020a].

Key idea 2: Programming with co-contexts using separation logic. The flipside of the co-contextual and nameless typed representation of bytecode, is that a lot of information is encapsulated in a single co-context composition (or merging) relation. This proof-relevant relation appears everywhere and is hard to manipulate by hand. To avoid this, we show that this relation forms a *proof-relevant separation algebra (PRSA)* [Rouvoet et al. 2020a], which allows us to build an embedded separation logic to abstract over co-contexts and their compositions. This separation logic allows us to implement an intrinsically typed version of the compiler in Fig. 1 with little manual proof work. The key result is summarized well by the type of the intrinsically typed freshLabel operation:

$$\text{freshLabel} \; : \; (\text{Binder } \psi \; * \; \text{Reference } \psi) \; \epsilon$$

Because labels are nameless there is no need for state, and thus freshLabel is not monadic. To ensure that output is well bound, we distinguish the two roles of labels in the types: *binding* and *reference* occurrences. The operation freshLabel constructs a pair of both a binding occurrence of type Binder ψ and a reference occurrence of type Reference ψ of the *same* label. The stack type ψ ensures that the two occurrence are used in type-compatible positions. The two occurrences are paired using a *separating conjunction* $*$. This hides the co-contexts, as well as the composition of these co-contexts that binds the reference occurrence to the binding occurrence. The ϵ on the outside is the *empty co-context*. The fact that the pair is typed using the empty co-context can be understood as this pair being internally well bound: it does not import any labels, nor export any labels for a user on the outside (i.e., the generated label really is fresh).

The rules of co-context composition prohibit the binding occurrence from being duplicated or discarded. This means that to the programmer, the binding occurrence returned by freshLabel behaves like a *linear value*. It can be passed around freely, but must eventually appear in the output of the compiler. This ensures that jumps in the output are intrinsically well bound.

Contributions. In this paper we present an approach to *intrinsic* verification of type correctness of compilers. After discussing the key challenges and the key ideas that we use to overcome these challenges (§2), we make the following technical contributions:

- In §3 we present a new nameless and co-contextual representation of typed global label binding. We formalize it using a generic, *proof-relevant*, ternary composition relation, that characterizes exports and imports and their possible interactions.
- In §4 we prove that this relation forms a proof-relevant separation algebra (PRSA) [Rouvoet et al. 2020a]—i.e., it is commutative, associative, and has a unit—which gives us a proof-relevant separation logic that abstracts over co-contexts and their compositions. This logic provides a high-level language for writing intrinsically typed programs that use nameless labels.
- In §5 we present a co-contextual typing of a subset of JVM bytecode with labels and jumps. We use our separation logic as a framework for specifying the typing rules, without explicitly talking about co-contexts and their compositions. We show that our co-contextual representation can be translated into code with absolute jumps instead of labels.
- In §6 we present the compilation of a small expression language with structured control-flow to JVM bytecode. To implement the compiler at the right level of abstraction with little manual proof work, we develop a linear writer monad for compiling with labels, implemented on top of our separation logic.
- In §7 we explain how the compilation pass fits in an intrinsically typed compiler backend that we implemented in Agda [Rouvoet et al. 2020b]. The backend also includes a source language transformation (local variable hoisting), a target language optimization (noop removal), and a transformation that eliminates labels in favor of instruction addresses.

We finish the paper with a discussion of related work (§8), and a conclusion (§9).

Although our separation logic completely abstracts over co-context compositions and avoids the work of proving bytecode well bound, we do not program the compiler entirely within the logic. The main reason for this is that dependent pattern matching in Agda can only be used to construct functions in Set. To systematically draw a line between the internal and external parts of the program, we will use a programming trick due to Poulsen et al. [2018], which involves explicit programming with monadic strength. This style of programming leaves room for improvement because some of the co-context composition witnesses show up in the compiler. We envision that the last step of hiding all co-context compositions requires a dependently typed meta language with better support for logical frameworks. Part of the contributions of this paper is to better understand the requirements of such a meta language.

Notation. The code in this paper is very close to the accompanying artifact [Rouvoet et al. 2020b]. We allow ourselves a few notational liberties. We omit top-level universally quantified variables, as well as universe levels in signatures. We present some types that have to be wrapped as records (to make type inference work) as plain types in the paper. Similarly we omit some type hints in places where Agda's type inference could not figure it out.

Raw syntax of bytecode

$$
\begin{array}{rcl}
\text{labels} & \ell & \\
\text{constants} & k & \\
\text{plain instructions} & i & := \quad \textbf{pop} \mid \textbf{push } k \mid \textbf{goto } \ell \mid (\ldots) \\
\text{labeled code points} & c & := \quad \textbf{plain } i \mid \textbf{labeled } \ell\ i \\
\text{bytecode} & b & := \quad \textbf{nil} \mid \textbf{cons } c\ b
\end{array}
$$

Contextually- and extrinsically-typed bytecode

T-Pop
$$\frac{}{I \vdash \textbf{pop} : \langle a :: \psi \rightsquigarrow \psi \rangle}$$

T-Push
$$\frac{k : a}{I \vdash \textbf{push } k : \langle \psi \rightsquigarrow a :: \psi \rangle}$$

T-Goto
$$\frac{I\ (\ell)\ =\ \psi_1}{I \vdash \textbf{goto } \ell : \langle \psi_1 \rightsquigarrow \psi_2 \rangle}$$

T-Plain
$$\frac{I \vdash i : \langle \psi_1 \rightsquigarrow \psi_2 \rangle \quad E = \emptyset}{E, I \vdash \textbf{plain } i : \langle \psi_1 \rightsquigarrow \psi_2 \rangle}$$

T-Labeled
$$\frac{I \vdash i : \langle \psi_1 \rightsquigarrow \psi_2 \rangle \quad E = \{\ell \mapsto \psi_1\}}{E, I \vdash \textbf{labeled } \ell\ i : \langle \psi_1 \rightsquigarrow \psi_2 \rangle}$$

T-Nil
$$\frac{E = \emptyset}{E, I \vdash \textbf{nil} : \langle \psi \rightsquigarrow \psi \rangle}$$

T-Cons
$$\frac{E_1, I \vdash c : \langle \psi_1 \rightsquigarrow \psi_2 \rangle \quad E_1 \sqcup E_2 \simeq E \quad E_2, I \vdash b : \langle \psi_2 \rightsquigarrow \psi_3 \rangle}{E, I \vdash \textbf{cons } c\ b : \langle \psi_1 \rightsquigarrow \psi_3 \rangle}$$

Contextually- and intrinsically-typed bytecode

StackTy = List Ty
LabelCtx = Map Label StackTy

data Instr : StackTy → StackTy → LabelCtx → Set **where**
  pop   : Instr $(a :: \psi)\ \psi\ I$
  push : Constant a → Instr $\psi\ (a :: \psi)\ I$
  goto : $I\ [\ \ell\ ] = \psi_1$ → Instr $\psi_1\ \psi_2\ I$

data Code : StackTy → StackTy → LabelCtx → LabelCtx → Set **where**
  plain   :   Instr $\psi_1\ \psi_2\ I$ → Code $\psi_1\ \psi_2\ \emptyset\ I$
  labeled : $\forall\ \ell$ → Instr $\psi_1\ \psi_2\ I$ → Code $\psi_1\ \psi_2\ (\ell \mapsto \psi_1)\ I$

data Bytecode : StackTy → StackTy → LabelCtx → LabelCtx → Set **where**
  nil   : Bytecode $\psi_1\ \psi_1\ \emptyset\ I$
  cons   : Code $\psi_1\ \psi_2\ E_1\ I$ → $(E_1 \sqcup E_2 \simeq E)$ → Bytecode $\psi_2\ \psi_3\ E_2\ I$ → Bytecode $\psi_1\ \psi_3\ E\ I$

Fig. 2. Named and contextually-typed bytecode.

2 INTRINSICALLY VERIFYING THAT LABELS ARE WELL BOUND

In this paper we present a novel nameless and co-contextual typing of bytecode and show that it is well suited for intrinsically typed compilation with labels. In this section we show the straightforward named and contextual typing of bytecode (§2.1), and why it falls short for intrinsically typed compilation (§2.2). We then explain our key ideas: nameless and co-contextual typing of bytecode (§2.3), and the use of separation logic for programming with it in a declarative manner (§2.4).

2.1 Named and Contextually-Typed Bytecode

Bytecode with labels can be described as a simple term language (Fig. 2). The plain instructions of this language can be typed using the judgment $I \vdash i : \langle \psi_1 \rightsquigarrow \psi_2 \rangle$, where the stack types

(lists of types) ψ_1 and ψ_2 express a precondition on the entry stack, and a postcondition to the next instruction in the bytecode sequence, respectively. To type jumps, we use a context I, which maps each referenced label to its entry stack type. The postcondition ψ_2 of **goto** is unconstrained because the next instruction in the sequence will not be executed after the goto has performed the jump. Instead, there is a premise $I\,(\ell)\ =\ \psi_1$ that ensures that the labeled target instruction ℓ—i.e., the actual next instruction of **goto** at runtime—has the right entry stack type.

Unlike plain instructions, code points Code can define labels. The judgments $E\,,\,I\ \vdash\ c\ :\ \langle\,\psi_1 \rightsquigarrow \psi_2\,\rangle$ for code points and $E\,,\,I\ \vdash\ b\ :\ \langle\,\psi_1 \rightsquigarrow \psi_2\,\rangle$ for bytecode therefore use an additional context E, which maps each binding occurrence of a label to its entry stack type. This context is *linear* so as to enforce that every label is defined *exactly once*. The linear behavior or the context E is visible in the leafs (T-Plain, T-Labeled, T-Nil), where we restrict E to the smallest possible context (i.e., no weakening), and in the nodes (T-Cons), where the condition $E_1 \sqcup E_2 \simeq E$ *separates* E in disjoint fashion among the sub-derivations (i.e., no contraction) [Walker 2005]. We refer to the disjoint separation of the linear context as the *disjointness condition* of label well-boundedness.

The typing judgment $E\,,\,I\ \vdash\ b\ :\ \langle\,\psi_1 \rightsquigarrow \psi_2\,\rangle$ itself does not enforce that every referenced label has a corresponding binder. Hence, in addition to a proof of the typing judgment, an *inclusion condition* $I \subseteq E$ needs to be proven at the top-level for the whole bytecode program. Equivalently, it suffices to prove a judgment $E\,,\,E\ \vdash\ b\ :\ \langle\,\psi_1 \rightsquigarrow \psi_2\,\rangle$ at the top level. After all, when $I \subseteq E$ holds, we can *weaken* a typing $E\,,\,I\ \vdash\ b\ :\ \langle\,\psi_1 \rightsquigarrow \psi_2\,\rangle$ to $E\,,\,E\ \vdash\ b\ :\ \langle\,\psi_1 \rightsquigarrow \psi_2\,\rangle$.

The first step towards proving compiler type-correctness *intrinsically* is to internalize the typing rules in the bytecode syntax. The result is an inductive type family Bytecode $\psi_1\ \psi_2\ E\ I$ (Fig. 2), indexed by the two stack types ψ_1 and ψ_2 of type StackTy [McKinna and Wright 2006], and the maps E and I of type LabelCtx. We leave the set of types Ty and the type of maps Map abstract. As usual, the intrinsically typed syntax integrates the syntax of bytecode with the typing judgment. Intrinsically typed label references are treated in a traditional manner: using context membership $I\,[\ \ell\ ]=\ \psi_1$, which say that I contains the label ℓ, and it has stack type ψ_1. Label definitions are treated analogously to the typing rules: via the linear context E. Consequently, the constructor **plain** enforces E to be the empty map $\emptyset$, whereas the constructor **labeled** sets E to be the singleton map $\ell\ \mapsto\ \psi_1$.

Again, we point out the two side conditions that ensure well-boundedness. The disjointness condition $E_1 \sqcup E_2 \simeq E$ in constructor **cons** of Bytecode ensures that labels are bound at most once. To ensure that every label that is referenced is also bound, it is necessary to also prove the inclusion condition $I \subseteq E$ for the whole program. We now show that these side conditions are a key obstacle that prevent us from implementing an intrinsically typed compiler without manual proof work.

2.2 Problem: Anti-Compositionality of Contextually Typed Bytecode

Let us take a look at a candidate signature for an intrinsically typed counterpart to the untyped monadic compiler in Fig. 1:

compile $:$ Exp $a\ \rightarrow\ \forall\ E_1\ \rightarrow\ (\exists\ \lambda\ E_2\ \rightarrow\ (E_1\ \#\ E_2) \times$ Bytecode $\psi\ (a :: \psi)\ E_2\ E_2)$

Here, Exp a is a type family for well-typed source expressions of type a. Similar to the untyped compiler, we thread a supply of fresh labels, represented concretely using the label context E_1 of already defined labels. The compiler then returns[1] the set of newly introduced labels E_2, which must be disjoint from E_1. We write disjointness $E_1\ \#\ E_2$, which is defined as $\exists\ \lambda\ E\ \rightarrow\ E_1 \sqcup E_2 \simeq E$. The returned bytecode Bytecode $\psi\ (a :: \psi)\ E_2\ E_2$ binds all the labels of E_2 and thus satisfies the

---

[1]We write $\exists\ \lambda\ x_1\ ...\ x_n\ \rightarrow\ A$ with $A\ :\ $ Set for the existential quantification over $x_1, ..., x_n$ (with inferred types) in A. Unlike universal quantification, the existential quantifiers is not a built-in type former in Agda, which is why it uses a lambda for the bound variables.

inclusion condition. The stack indices of Bytecode ψ $(a :: \psi)$ E_2 E_2 (i.e., the pre stack type ψ and post stack type $a :: \psi$) enforce the stack invariant of compiled expressions.

Although it is possible to implement the function compile with the given signature, such implementations do not have the declarative appeal of the untyped compiler in Fig. 1. Compilation of expressions like **if** c **then** e_1 **then** e_2 require multiple recursive invocations of the compiler, which requires threading of the label context, as well as manual concatenation of the outputs. The latter comes with proof obligations. In particular, we have to weaken the imports of typed bytecode to the set of all generated labels. We also have to prove the disjointness conditions required for concatenation. This results in a definition that is bloated with proof terms.

One may wonder if we can hide the threading of the label context, and the concatenation of the output in a monad, as we did in the untyped compiler. This would require us to type the monadic operations (return, bind, freshLabel, tell, and attach) so that we can implement the compiler compositionally with little manual proof work. In what follows, we will argue that contextually typed bytecode is not well suited for that, because the two conditions for label well-boundedness are inherently anti-compositional, whole program properties.

Proving the inclusion condition. The first problem is that the inclusion condition of bytecode only holds for whole programs. That is, while it holds for the output of compile, it does not hold for the individual monadic operations that we wish to use. For example, the output written by tell [**goto** l], which has type Bytecode ψ ψ $\emptyset$ $(\ell \mapsto \psi)$, does not satisfy the inclusion condition, because $(\ell \mapsto \psi) \not\subseteq \emptyset$. It thus appears inevitable that the inclusion condition is proven *extrinsically* in the implementation of expression compilation, because it is something that only holds at the level of whole programs, and not at the level of the individual monadic operations.

Proving the disjointness condition. The second problem is related to the disjointness condition of well-boundedness. In the untyped compiler in Fig. 1, disjointness is morally ensured through principled use of freshLabel. An intrinsically typed version of freshLabel will thus have to provide some evidence of freshness: a proof $(\ell \mapsto \psi) \# E_1$ that the returned label ℓ is disjoint from already bound ones E_1. This evidence is required when invoking attach ℓ. The problem, however, is that the freshness evidence is not stable. Consider, for example, the following untyped compilation:

do $\ell \leftarrow$ freshLabel; compile e; attach ℓ (return ())

Assume it is indeed the case that before generating ℓ the set of already bound labels was E_1, and we get the evidence [ℓ] # E_1. By the time we want to attach ℓ, an additional (existentially quantified) number of labels E_2 have been bound by the compilation of e. Consequently, we will need evidence [ℓ] # $(E_1 \cup E_2)$ to be able to safely attach ℓ. Proving this requires explicit reasoning about disjointness, combining the evidence returned by freshLabel with the evidence returned by the recursive invocation to compile.

These problems are symptomatic of the anti-compositional nature of typing label binding in bytecode. To some extent, this was to be expected: label names have to be globally well bound and unique. Contextual bytecode typing enforces this as a whole-program property. The monadic compiler, however, is mostly manipulating *partial* bytecode programs, as it constructs its output piecewise. Hence, we are facing the question how we can *generalize* whole program typing to partial programs. The only way to do this is via *side conditions* that express how individual operations contribute to a proof that exceeds their local scope: the proof of the top-level conditions.

2.3 Key Idea 1: Nameless Co-Contextual Bytecode

Instead of further pursuing this generalization of contextual typing, we propose a *co-contextual* reformulation of bytecode where labels are *nameless*. Using our nameless encoding we avoid the

Co-Pop
$$\frac{\boxed{I \;=\; []}}{I \;\vdash\; \textbf{pop} \;:\; \langle\, a :: \psi \rightsquigarrow \psi \,\rangle}$$

Co-Push
$$\frac{k \;:\; a \qquad \boxed{I \;=\; []}}{I \;\vdash\; \textbf{push}\; k \;:\; \langle\, \psi \rightsquigarrow a :: \psi \,\rangle}$$

Co-Goto
$$\frac{\boxed{I \;=\; [\,\psi_1\,]}}{I \;\vdash\; \textbf{goto} \;:\; \langle\, \psi_1 \rightsquigarrow \psi_2 \,\rangle}$$

Co-Plain
$$\frac{I \;\vdash\; i \;:\; \langle\, \psi_1 \rightsquigarrow \psi_2 \,\rangle \qquad E \;=\; []}{E \Updownarrow I \;\vdash\; \textbf{plain}\; i \;:\; \langle\, \psi_1 \rightsquigarrow \psi_2 \,\rangle}$$

Co-Labeled
$$\frac{I \;\vdash\; i \;:\; \langle\, \psi_1 \rightsquigarrow \psi_2 \,\rangle \qquad E \;=\; [\,\psi_1\,]}{E \Updownarrow I \;\vdash\; \textbf{labeled}\; i \;:\; \langle\, \psi_1 \rightsquigarrow \psi_2 \,\rangle}$$

T-Nil
$$\frac{\boxed{K \;=\; \epsilon}}{\boxed{K} \;\vdash\; \textbf{nil} \;:\; \langle\, \psi \rightsquigarrow \psi \,\rangle}$$

Co-Cons
$$\frac{\boxed{K_1} \;\vdash\; c \;:\; \langle\, \psi_1 \rightsquigarrow \psi_2 \,\rangle \qquad \boxed{K_1 \;\bullet\; K_2 \;\simeq\; K} \qquad \boxed{K_2} \;\vdash\; b \;:\; \langle\, \psi_2 \rightsquigarrow \psi_3 \,\rangle}{\boxed{K} \;\vdash\; \textbf{cons}\; c\; b \;:\; \langle\, \psi_1 \rightsquigarrow \psi_3 \,\rangle}$$

Fig. 3. Co-contextually typed bytecode.

difficulties that we sketched in §2.2. The main reason for this is that a nameless representation rules out any sort of *accidental overlap* between labels. That is, unless a function gets passed a label explicitly, it has no means to get hold of that label. As a consequence, it is unnecessary for compile to provide evidence that it does not bind the labels that we intent to attach later. It could not possibly do this, because we have not informed it about the existence of these labels. As we will see in §2.4, our nameless representation makes it possible to assign strong types to the monadic operations on bytecode that intrinsically ensure label well-boundedness.

The means by which we obtain this nameless representation is via a *co-contextual* typing of bytecode with a *proof-relevant* notion of co-context merging. We introduce this idea first using typing rules, so as to make it easy to compare to the contextual typing rules in §2.1. We say that a typing rule is co-contextual [Erdweg et al. 2015; Kuci et al. 2017] if instead of receiving a context, it produces a co-context. Co-contexts must pertain to the term at hand and contain no information that is irrelevant to it. This is useful in dependently typed programming, which benefits from expressing invariants as *local knowledge* [McBride 2014]. For bytecode, co-contexts are pairs of an *export* context E and *import* context I, which we call *interfaces* K, and write $E \Updownarrow I$. The roles of E and I are analogous to the E and I in the contextual rules. However, they are both *lists* of label types (rather than maps), since we have done away with names. Additionally, I is constrained to be the smallest possible in the same way as the linear context E, which is naturally in a co-contextual style. For example, nil has context $\epsilon \;=\; [] \Updownarrow []$. The resulting rules are summarized in Fig. 3, highlighting the differences with a shaded background.

How does the nameless representation work? The goto and labeled operations no longer contain labels, so how does one know where to jump to? The key idea—inspired by McBride's co-de-Bruijn representation [McBride 2018]—is that type derivations (which in the intrinsically typed version will be part of the syntax) contain that information. In particular, the ternary relation $K_1 \bullet K_2 \simeq K$ for interface composition (or interface separation, depending on the perspective), is *proof relevant*. This means that proofs of $K_1 \bullet K_2 \simeq K$ are values that contain precise information describing the way reference occurrences (appearing in I) are related to their binding counterparts (in E).

Why does this address the problems with the contextual typing from §2.1? As we already emphasized, by going nameless we no longer have accidental overlap between labels, Therefore, unlike the linear context separation relation $E_1 \sqcup E_2 \simeq E$ (appearing in the contextual typing rule T-Cons), the co-contextual counterpart $K_1 \bullet K_2 \simeq K$ (in Co-Cons) is not a proof *obligation*. Rather, it represents a *choice* how to relate the labels in K_1 with the labels in K_2. There is one trivial

proof of $K_1 \bullet K_2 \simeq K$: the labels in K_1 and K_2 are independent, which happens when combining two recursive calls of the compiler. This choice requires no further evidence. In other words: unlike in the contextual formulation, avoiding accidental label binding requires *no* cooperation from the compiler writer. Apart from the trivial proof, one can construct non-trivial proofs of $K_1 \bullet K_2 \simeq K$ that relate references in K_1 with binders in K_2, and vice versa.

Another important aspect of interface composition $K_1 \bullet K_2 \simeq K$, is that binding and reference occurrences *cancel* each other out. This formalizes the idea that the reference occurrences are also obligations that are fulfilled by corresponding binding occurrences. All copies of the same reference occurrence are canceled out by a single binding occurrence. Because of this, labels that are used *and* defined in a fragment of bytecode, and are not used outside of this fragment, do *not appear in the exports of that fragment*. This effectively achieves scoping of labels, but without adding more structure to the bytecode that is not really present in the language. This *logical scoping* of labels allows us to prove the inclusion condition locally, rather than as a whole-program property.

2.4 Key Idea 2: Programming with Co-Contexts Using Separation Logic

The question now becomes how we write programs using labels when they are nameless. As we will see in §3, the actual definition of the proof-relevant interface composition relation $K_1 \bullet K_2 \simeq K$ is complicated. Certainly, if we have to manually work with the definition of that relation, then we fail to accomplish our goal of writing proof-free compilers. This brings us to the remaining idea: abstracting over interfaces and their composition using an embedding of *proof-relevant separation logic*, and treating binding occurrences as *linear values* in a compiler.

To define the compiler, we need to define a type family that internalizes the co-contextual typing rules in the nameless bytecode syntax (§5). The actual definition can be found in §5, but for now, we are content with only defining the syntax of intrinsically typed *labels*. We define Binder for binding occurrences, and Reference for reference occurrences:

> **data** Binder : StackTy → Intf → Set **where**
>   **binder** : Binder ψ ([ψ] ⇑ [])
> **data** Reference : StackTy → Intf → Set **where**
>   **reference** : Reference ψ ([] ⇑ [ψ])

The stack type ψ indicates the type of the label occurrences. A binding occurrence with type ϕ must be attached to an instruction that expects a stack typed ψ. A reference occurrence with type Φ must be used by a jump that leaves behind a stack typed ψ. The type Intf is the representation of interfaces and has constructor _⇑_. The interface of the binding occurrence consists of the singleton export of ψ, whereas the reference occurrence consists of the singleton import of ψ.

We can now give the type of the touchstone of our nameless representation of labels:

> freshLabel : ($\exists \lambda K_1\ K_2$ → Binder ψ K_1 × ($K_1 \bullet K_2 \simeq \epsilon$) × Reference ψ K_2)

That is, freshLabel is a *constant* pair, of a binding and reference occurrence. It is constant, because there is no need to invent a unique name. The binding relation is established by the glue that sits in between: the composition of the interfaces $K_1 \bullet K_2 \simeq \epsilon$. The singleton export and import cancel each other out, so that the pair itself has the empty interface ϵ. This means that this compound object is internally well bound: it does not import or export any labels from the surroundings. This naturally means that the occurrences are fresh with respect to any other label in the surroundings.

The final step is to realize that we can write this more succinctly if we abstract everywhere over interfaces and use the connectives of *separation logic*. That is, we write the signature of freshLabel using a *proof relevant separating conjunction* ∗ [Rouvoet et al. 2020a] as follows:

> freshLabel : (Binder ψ ∗ Reference ψ) ϵ

Here, the separating conjunction $*$ will be defined roughly as follows:

$$\_*\_ \ : \ (\mathsf{Intf} \ \to \ \mathsf{Set}) \ \to \ (\mathsf{Intf} \ \to \ \mathsf{Set}) \ \to \ \mathsf{Intf} \ \to \ \mathsf{Set}$$
$$P \ * \ Q \ = \ \lambda \ K \ \to \ (\exists \ \lambda \ K_1 \ K_2 \ \to \ P \ K_1 \times (K_1 \ \bullet \ K_2 \simeq K) \times Q \ K_2)$$

In this way, separation logic provides a *high-level* language in which we can express the types and implementation of operations on the co-contextually typed bytecode, at a suitable level of abstraction. From this perspective, label interfaces are a proof-relevant resource.

In the next section (§3) we discuss how interface composition is a *proof-relevant separation algebra* (PRSA) [Rouvoet et al. 2020a]. This gives us the well-behaved model of separation logic on predicates over interfaces Intf (§4). We use this high-level language as a logical framework for defining the intrinsically typed, co-contextual version of bytecode in §5, and also to type all operations on bytecode and nameless labels. As a glimpse forward, we now end this section with the signature of the typed compiler (slightly simplified from the real definitions in §6):

$$\mathsf{Compiler} \ \psi_1 \ \psi_2 \ A \ = \ \mathsf{Bytecode} \ \psi_1 \ \psi_2 \ * \ A$$

return	:	$(A \ {-\!\!*} \ \mathsf{Compiler} \ \psi \ \psi \ A) \ \epsilon$
bind	:	$((A \ {-\!\!*} \ \mathsf{Compiler} \ \psi_2 \ \psi_3 \ B) \ {-\!\!*} \ \mathsf{Compiler} \ \psi_1 \ \psi_2 \ A \ {-\!\!*} \ \mathsf{Compiler} \ \psi_1 \ \psi_3 \ B) \ \epsilon$
tell	:	$(\mathsf{Bytecode} \ \psi_1 \ \psi_2 \ {-\!\!*} \ \mathsf{Compiler} \ \psi_1 \ \psi_2 \ \mathsf{Emp}) \ \epsilon$
attach	:	$(\mathsf{Binder} \ \psi_1 \ {-\!\!*} \ \mathsf{Compiler} \ \psi_1 \ \psi_2 \ A \ {-\!\!*} \ \mathsf{Compiler} \ \psi_1 \ \psi_2 \ A) \ \epsilon$
compile	:	$\mathsf{Exp} \ a \ \to \ \mathsf{Compiler} \ \psi \ (a :: \psi) \ \mathsf{Emp} \ \epsilon$

If you squint your eyes so that the separating conjunction $*$ becomes a normal product, the magic wand $-\!\!*$ becomes a normal function arrow, and Emp becomes Unit, then the monad is "just" an indexed writer monad with the usual return, bind, and tell. The indexing with stack types ψ_1 and ψ_2 is used to type the bytecode output. The attach operation takes a binding occurrence of a label, which is made apparent in its type. As visible in the above types, the top-level embedding of the objects of our separation logic of type $\mathsf{Intf} \ \to \ \mathsf{Set}$ is achieved by supplying the empty interface ϵ.

The separation logic connectives help us abstract over interfaces and their compositions. This works uniformly for the low-level operations of compilation and for expression compilation, despite the fact that only expression compilation really preserves label well-boundedness. We will show that the same logic is useful to define the typed syntax, and non-monadic functions over syntax.

3 A MODEL OF NAMELESS CO-CONTEXTUAL BINDING

We present a model of nameless co-contextual global binding. Its elements K are *binding interfaces*, which abstractly describe the label binders and label references of a syntax fragment. Interfaces are composed using a ternary proof-relevant relation $K_1 \ \bullet \ K_2 \simeq K$. We first present the model using a handful of examples that demonstrate its key features (§3.1), and then define it formally (§3.2). Finally, we explain why proof relevance is essential, by considering the translation from labels to absolute addresses (§3.3).

3.1 Interfaces Composition Exemplified

We define *label interfaces* $K \ : \ \mathsf{Intf}$ as pairs of label typing contexts LabelCtx:

```
record Intf : Set where              StackTy  = List Ty
  constructor _↧_                     LabelCtx = List StackTy
  field
    exp : LabelCtx
    imp : LabelCtx
```

The code declares Intf as a record with constructor $(E \Updownarrow I)$, and projections exp and imp, which describe the label binders (the *exports*) and label references/jumps (the *imports*), respectively. Recall from §2.3 that the fact that we use a nameless representation of label binding is visible in two ways. First, label contexts LabelCtx are mere lists.[2] Second, the relation $K_1 \bullet K_2 \simeq K$ for composition of interfaces K_1 and K_2 into K is proof relevant—i.e., proofs of $K_1 \bullet K_2 \simeq K$ are values that choose between options of relating the labels in K_1 to those in K_2. Before we formally define this relation (§3.2), we illustrate its key features through a number of examples.

The simplest way to compose interfaces—which is applicable for any choice of operand interfaces—is to take the disjoint union of the imports and exports of the interface:

$$(E_1 \Updownarrow I_1) \bullet (E_2 \Updownarrow I_2) \simeq ((E_1 \sqcup E_2) \Updownarrow (I_1 \sqcup I_2)) \tag{3.1}$$

In this case there is no interaction between the binding of the parts, and their label use is completely disjoint. A concrete instance of the above composition would be to compose two interfaces that both have an import (i.e., a label reference) of type ϕ:

$$([] \Updownarrow [\ \phi\ ]) \bullet ([] \Updownarrow [\ \phi\ ]) \simeq ([] \Updownarrow [\ \phi\ , \phi\ ]) \tag{3.2}$$

The interfaces $([] \Updownarrow [\ \phi\ ])$ could be assigned to two bytecode sequences that both contain a single jump instruction with target stack type ϕ. By taking the disjoint union, we express that these jump instructions refer to disjoint label binders. If we wish to express that these jump instructions refer to the same binder, then the composition relation can contract them. For example:

$$([] \Updownarrow [\ \phi\ ]) \bullet ([] \Updownarrow [\ \phi\ ]) \simeq ([] \Updownarrow [\ \phi\ ]) \tag{3.3}$$

Equation 3.2 and Equation 3.3 show that the relation $K_1 \bullet K_2 \simeq K$ is not functional—there is a choice whether to contract imports in K_1 or K_2, or not. Apart from not being functional, the relation $K_1 \bullet K_2 \simeq K$ is also proof relevant. The different proofs of $K_1 \bullet K_2 \simeq K$ are values that represent the different choices of composing K_1 and K_2. For example, consider a composition of two interfaces where the left has one import of type ϕ, and the right and composite interfaces have two imports of type ϕ:

$$([] \Updownarrow [\ \phi\ ]) \bullet ([] \Updownarrow [\ \phi\ , \phi\ ]) \simeq ([] \Updownarrow [\ \phi\ , \phi\ ]) \tag{3.4}$$

The interface $([] \Updownarrow [\ \phi\ ])$ on the left could be assigned to bytecode that contains a single jump instruction with target stack type ϕ, while the interface $([] \Updownarrow [\ \phi\ , \phi\ ])$ on the right could be assigned to bytecode that contains two jump instructions with target stack type ϕ. We can now contract the import on the left in two ways—do we wish to contract it with the first or the second import on the right? Since we consider the composition relation to be proof relevant, there are two proofs of $([] \Updownarrow [\ \phi\ ]) \bullet ([] \Updownarrow [\ \phi\ , \phi\ ]) \simeq ([] \Updownarrow [\ \phi\ , \phi\ ])$ corresponding to this choice.

We have seen that the relation $K_1 \bullet K_2 \simeq K$ can be used to contract imports in K_1 and K_2, which corresponds to expressing different jump instructions refer to the same target binder. Contraction of exports is impossible, as it would not make sense to express that two binders should become the same target. Thus, for example, we do *not* have:

$$([\ \phi\ ] \Updownarrow []) \bullet ([\ \phi\ ] \Updownarrow []) \simeq ([\ \phi\ ] \Updownarrow []) \tag{3.5}$$

Given the sub-structural nature of composition, we conclude that exports are treated essentially as *linear*, and imports as *relevant*. This agrees with our intent to formulate co-contextual typings.

---

[2]In some sense label contexts are really multisets or bags, because everything is stable under permutation of the lists. The proof term of composition does depend on the order of elements however.
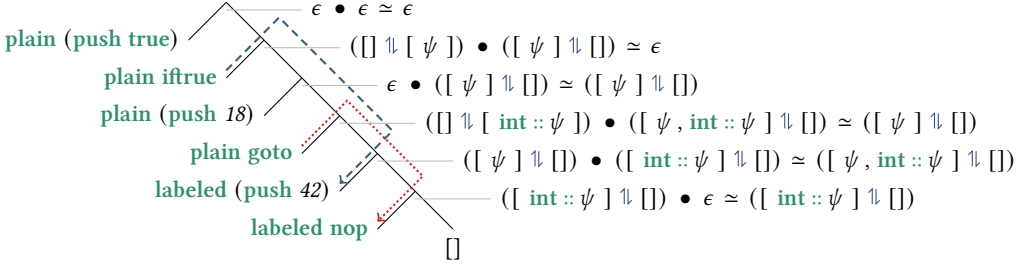
Fig. 4. Interface compositions in the result of compilation of **if true then** *42* **else** *18*. For every composition $K_1 \bullet K_2 \simeq K$, the interfaces K_1 and K_2 describe the imports and exports of the head and tail of that node respectively, whereas K does the same for the entire node. Consequently, K always matches the interface of the tail of the node above.

We now turn to binding an import to an export. Using composition $K_1 \bullet K_2 \simeq K$, we may bind an import from K_1 to an export from K_2 of the same type, and vice versa. For example, we can bind an import typed ϕ from the left, using an export typed ϕ from the right:

$$([\,] \mathbin{\Updownarrow} [\,\phi\,,\psi\,]) \bullet ([\,\phi\,] \mathbin{\Updownarrow} [\,]) \simeq ([\,\phi\,] \mathbin{\Updownarrow} [\,\psi\,]) \tag{3.6}$$

The interface $([\,] \mathbin{\Updownarrow} [\,\phi\,,\psi\,])$ on the left could be assigned to bytecode that contains two jump instructions to label references of types ϕ and ψ, while the interface $([\,\phi\,] \mathbin{\Updownarrow} [\,])$ on the right could be assigned to bytecode that contains no (unbound) jump instructions, but a binding occurrence of type ϕ. The composition expresses that the reference occurrence of the label of type ϕ on the left is bound to the binding occurrence on the right. Because the import of ϕ is fulfilled—i.e., the jump instruction to ϕ has been bound—ϕ does not reappear in the imports of the composite.

When binding an import using an export, there is a choice whether we want to use the export again or not. In Equation 3.6, the export ϕ remains available in the composition, which means that another jump instruction can target it as well. Instead, we could decide to *hide* the export ϕ:

$$([\,] \mathbin{\Updownarrow} [\,\phi\,,\psi\,]) \bullet ([\,\phi\,] \mathbin{\Updownarrow} [\,]) \simeq ([\,] \mathbin{\Updownarrow} [\,\psi\,]) \tag{3.7}$$

Hiding an export is only allowed if it has been bound by an import. In other words, if we choose *not* to bind an import using an export, then we cannot hide the export. This ensures that the rules are truly co-contextual by keeping the label contexts tight (and also rules out unused label declarations). This ensures that the model has no mysterious compositions with the unit ϵ, like:

$$\epsilon \bullet ([\,\phi\,] \mathbin{\Updownarrow} [\,]) \simeq \epsilon \tag{3.8}$$

We make the properties of identity compositions more precise in the next section.

The hiding gives rise to *logical scopes*, which are not present in the syntactical structure, but are present in the proof-relevant binding model. In Fig. 4 we see how the various features of the composition relation can be used to type the bytecode corresponding to the expression **if true then** *42* **else** *18*. The tree represents the list-like bytecode abstract syntax, where every node is annotated with the interface composition $K_1 \bullet K_2 \simeq K$. Since we use a nameless representation, the only way to tell which label goes where (visualized using the blue and red arrow) is by examining the proofs that witness the composition $K_1 \bullet K_2 \simeq K$. The labels are only in scope in the parts of the tree that the dotted lines traverse. Most notable, they can be in scope in the *spine* of the tree, without being in scope of some of the instructions that are attached to it. The figure also shows how labels are hidden from instructions higher in the sequence using export hiding: when we bind a labeled instruction to a **goto** (respectively, **iftrue**), the corresponding export of type **int** :: ψ (respectively, ψ) is hidden, because there are no further uses elsewhere.

3.2 Interface Composition Defined Formally

We now present the formal definition of interface composition $K_1 \bullet K_2 \simeq K$. The formal definition must clarify (1) which imports from K_1 are bound by exports of K_2 and vice versa, and (2) which imports and exports from K_1 and K_2 are exposed by the composite K. We take care to specify this in a way that ensures that each bound import resolves *unambiguously* to a single export. In particular, this means that each import that is bound to an export *cannot* reappear in the composite imports (see Equation 3.6). Conversely, each import that is not bound *must* appear in the import list of the composite K. In addition, exports are treated *relevantly*: if K_1 (respectively, K_2) exports a binder that is not used to bind an import of K_2 (respectively, K_1), then it *must* reappear in the exports of the composite K. But, if exports in either K_1 (respectively, K_2) are bound by an input in K_2 (respectively, K_1) then they *can* be hidden in K.

To define the interface composition $K_1 \bullet K_2 \simeq K$, we use two ternary relations of a similar flavor that specify disjoint context separation and context separation with overlap, respectively:

$$\_ \sqcup \_ \simeq \_ \; : \; \mathsf{LabelCtx} \;\rightarrow\; \mathsf{LabelCtx} \;\rightarrow\; \mathsf{LabelCtx} \;\rightarrow\; \mathsf{Set} \quad \text{-- Disjoint context separation}$$
$$\_ \cup \_ \simeq \_ \; : \; \mathsf{LabelCtx} \;\rightarrow\; \mathsf{LabelCtx} \;\rightarrow\; \mathsf{LabelCtx} \;\rightarrow\; \mathsf{Set} \quad \text{-- Context separation with overlap}$$

Interface composition is now defined as follows:

```
data Binds : LabelCtx → LabelCtx → LabelCtx → LabelCtx → Set where
   binds :  (I⁻  ⊔  B ≃ I)
         → (E⁻  ∪  B ≃ E)
         → Binds E I E⁻ I⁻
data _ • _ ≃ _ : Intf → Intf → Intf → Set where
   comp :  Binds E₁ I₂ E₂⁻ I₂⁻
        → Binds E₂ I₁ E₁⁻ I₁⁻
        → (E₁⁻ ⊔ E₂⁻ ≃ E)  → (I₁⁻ ∪ I₂⁻ ≃ I)
        → ((E₁ ⇑ I₁) • (E₂ ⇑ I₂) ≃ (E ⇑ I))
```

The composition relation expresses that the composite $(E \Uparrow I)$ of $(E_1 \Uparrow I_1)$ and $(E_2 \Uparrow I_2)$ is obtained by (1) binding some of the imports I_2 to exports E_1, resulting in "leftovers" I_2^- and E_2^-, and symmetrically binding certain I_1 to E_2, resulting in "leftovers" I_1^- and E_1^-, (2) adding up the leftover imports and exports to obtain E and I. Exports are combined without overlap (i.e., using $\sqcup$), while imports are combined *with overlap* (i.e., using $\cup$), implementing import contraction. The intuition behind the asymmetry between the use of disjoint separation for imports and separation with overlap for exports comes from the fact that labels should be bound only once, whereas they can be referenced multiple times.

In order to express which imports are bound to which exports, we define the auxiliary relation Binds $E \; I \; E^- \; I^-$. This relation specifies that E^- and I^- consist of the "leftover" exports and imports after *binding* some of the imports I to the exports E. It disjointly separates the imports I into those to be bound B, and those that are leftover I^-. As shown in the definition of $\_ \bullet \_ \simeq \_$, the leftover imports will have to reappear in the imports of the composite interface. The bound imports B are also subtracted from the exports E to get the leftover exports E^-. There can be overlap between E and E^-, giving the option to re-export used exports from the composite, as shown in Equation 3.6. Or, conversely, we only have the choice to hide an export from an interface if it has been bound by an import, thus precluding the composition in Equation 3.8.

3.3 From Nameless Labels to Addresses

Since our binding model uses nameless label binders and references, it is essential that the interface composition relation $\_ \bullet \_ \simeq \_$ is proof relevant. That is, labels are given a meaning not by the mere fact that there *exist* witnesses for interface composition in all nodes of the typing derivation, but by the *specific* witnesses that have been used to construct the typing derivation. Although cryptic in its form, the composition relation really lays out the binding paths visualized in Fig. 4.

The proof relevance becomes most apparent in the last pass of our compiler backend (§7), where our nameless representation is translated into a representation with jumps to absolute addresses. This transformation works by computing an environment that assigns an address to every label in the imported label context I. This environment is split and recombined along the witnesses of the interface composition in each node of the typing derivation. This way, jumps follow the path laid out by the witnesses of the interface composition relation.

4 PROGRAMMING WITH CO-CONTEXTS USING SEPARATION LOGIC

Interface composition models nameless label binding using the proof-relevant interface composition relation $\_\bullet\_\simeq\_$. Since the definition of the interface composition relation is complex, one would be hard pressed to manually construct witnesses each time an interface composition needs to be established, e.g., in the premises of the typing rules in Fig. 4. To write compilers with little manual proof work, we thus want to avoid manual construction of such witnesses in functions that transform co-contextually typed labels and bytecode. Instead, we want to use labels as symbolic values, similar to the way labels are used in the untyped compiler in Fig. 1.

To accomplish this, we abstract over interfaces and their compositions by proving that interface composition is a proof-relevant separation algebra (PRSA) [Rouvoet et al. 2020a] (§4.1), by which we obtain a proof-relevant separation logic on predicates over interfaces (§4.2). Separation logic provides a substructural specification and programming language, shallowly embedded in Agda (§4.3) in which we specify typed bytecode (§5), and specify and implement functions on bytecode, without manually constructing interface composition witnesses (§6).

4.1 Proof-Relevant Separation Algebras

A proof-relevant separation algebra (PRSA) [Rouvoet et al. 2020a] is a commutative, partial monoid on a carrier A. The unit is written ϵ, and the operation of the monoid is described by a ternary proof-relevant relation $\_\bullet\_\simeq\_$. The laws of a PRSA are given with respect to a given equivalence relation $\approx$ on the carrier A:

$\bullet\text{-id}^{\mathsf{l}}$: $(\epsilon \bullet a \simeq a)$
$\bullet\text{-id}^{-\mathsf{l}}$: $(\epsilon \bullet a \simeq b) \rightarrow a \approx b$
$\bullet\text{-comm}$: $(a \bullet b \simeq c) \rightarrow (b \bullet a \simeq c)$
$\bullet\text{-assoc}$: $(a \bullet b \simeq ab) \rightarrow (ab \bullet c \simeq abc) \rightarrow \exists \lambda\, bc \rightarrow (a \bullet bc \simeq abc) \times (b \bullet c \simeq bc)$

In addition, the relation $\_\bullet\_\simeq\_$ must respect the equivalence relation $\approx$ in all three positions. In contrast to earlier work on (variants of) separation algebras, for example [Calcagno et al. 2007; Dockins et al. 2009; Jung et al. 2018], the operation of a PRSA is not required to be functional:

$\oslash\; \bullet\text{-func}$: $a \bullet b \simeq c \rightarrow a \bullet b \simeq d \rightarrow c \approx d$

This reflects that PRSAs formalize proof-relevant relations that may offer different ways of putting the same elements together.

Both disjoint context separation $\_ \sqcup \_ \simeq \_$ and context separation with overlap $\_ \cup \_ \simeq \_$ from §3.2 are instances of PRSAs with the empty list [] as the unit, and proof-relevant list permutations as

the equivalence $\approx$. Interface composition as defined in §3.2 is also a PRSA, with the empty interface $\epsilon = [] \Updownarrow []$ as the unit, and pointwise proof-relevant list permutations as the equivalence $\approx$.

4.2 Proof-Relevant Separation Logic

Every PRSA induces a separation logic on proof-relevant predicates over the carrier A. The index A can be thought of as the amount of *resource* owned (not necessarily exclusively) by an inhabitant of the predicate. In this section we construct this model of separation logic in the usual way [O'Hearn and Pym 1999], and then show in §4.3 how it can be used to specify and implement functions in a dependently typed language [Rouvoet et al. 2020a].

We construct a separation logic over an arbitrary PRSA with carrier A, relation $\_ \bullet \_ \simeq \_$, and unit ϵ. In this paper we instantiate this logic for various concrete PRSAs. Hence, whenever the PRSA is unambiguously determined, we will use the constructions from this section in an overloaded fashion. The *separating conjunction* $P * Q$, and predicates Own a and Emp that witness ownership of a resource a and the unit resource ϵ, respectively, are defined in Agda as follows:

record $\_*\_$ $(P \; Q : \mathsf{Pred} \; A) \; (a : A) : \mathsf{Set}$ **where**
  constructor $\_\bullet\langle\_\rangle\_$
  field
    $\{a_l \; a_r\} : A$
    $px : P \; a_l$
    $sep : a_l \bullet a_r \simeq a$
    $qx : Q \; a_r$

$\mathsf{Pred} \; A = A \to \mathsf{Set}$

$\mathsf{Own} : A \to \mathsf{Pred} \; A$
$\mathsf{Own} \; a = \lambda \; b \to b \equiv a$

$\mathsf{Emp} : \mathsf{Pred} \; A$
$\mathsf{Emp} = \mathsf{Own} \; \epsilon$

We define the separating conjunction $P * Q$ as a record to hide the existential quantification over the two resources a_l and a_r. The constructor of the $*$ is written $px \; \bullet\langle \; sep \; \rangle \; qx$, and will appear when we construct or destruct a separating conjunction.

Besides the separation logic connectives, we also make use of the ordinary (pointwise) connectives on predicates, in particular the pointwise arrow $P \Rightarrow Q$ and the universal closure $\forall[\; P \;]$. As a convenient alternative to $\forall[\; \mathsf{Emp} \Rightarrow P \;]$, we also define the ϵ-*closure* $\epsilon[\; P \;]$:

$(P \Rightarrow Q) = \lambda \; a \to P \; a \to Q \; a$
$\forall[\; P \;] \quad\;\; = \forall \; \{a\} \to P \; a$
$\epsilon[\; P \;] \quad\;\; = P \; \epsilon$

Our separation logic enjoys the usual laws of separation logic, for example:[3]

$*\text{-swap} \quad : \forall[\; (P * Q) \quad\quad\;\; \Rightarrow (Q * P) \;] \qquad *\text{--}\mathsf{id}^\mathsf{l} : \forall[\; P \Rightarrow P * \mathsf{Emp} \;]$
$*\text{-assoc}_\mathsf{r} \; : \forall[\; (P * Q) * R \Rightarrow P * (Q * R) \;] \qquad *\text{--}\mathsf{id}^{\mathsf{-l}} : \forall[\; \mathsf{Emp} * P \Rightarrow P \;]$
$*\text{-rotate}_\mathsf{r} : \forall[\; P * Q * R \quad\;\; \Rightarrow R * P * Q \;]$

The adjoint of the separating conjunction $*$ is the *separating implication* or *magic wand* $P \multimap Q$:

$P \multimap Q = \lambda \; a_l \to \forall \; \{a_r \; a\} \to a_l \bullet a_r \simeq a \to P \; a_r \to Q \; a$

The resource a_l in an inhabitant of the wand of type $(P \multimap Q) \; a_l$ can be thought of as the resources in Q minus the resources in P—i.e., the resources in the closure. Since constructing wands in Agda is unpleasant (unless done in point-free style), we avoid them where possible. Instead, we make use of the fact that there exists an equivalence between resource-less wands and the $\forall$-quantified pointwise arrow: $\epsilon[\; P \multimap Q \;]$ iff $\forall[\; P \Rightarrow Q \;]$. Because the latter is directly defined in terms of Agda's function type, it is much more pleasant to work with and we prefer it wherever possible.

---

[3]To use the inverse identities (e.g., $*\text{--}\mathsf{id}^{\mathsf{-l}}$), the predicate P needs to respect the equivalence relation. Formally, this must be proven for all objects of the logic. The library relaxes this, using a type class constraint where necessary.

4.3 Programming with Separation Logic

While separation logic is conventionally used to reason about the dynamic behavior of heap-manipulating programs, we use it in a different way. Instead of using it to prove program properties extrinsically, we use it to manipulate intrinsically typed syntax in Agda. Moreover, instead of using it for dynamic program properties, we use it to capture static properties. Hence, contrary to the conventional use of separation logic where one abstracts over dynamic resources like heaps, we abstract over static resources representing label contexts and interfaces.

A key part of the implementation of our compiler is a writer monad that collects the generated bytecode. Compared to the ordinary writer monad, there are two twists. First, our writer monad is written in the category of separation logic predicates instead of the category of plain types, so that the handling of label binding is encapsulated. This twist is based on the work of Rouvoet et al. [2020a], who have shown that it is possible to use separation logic to write monadic interpreters for linear languages. Second, our writer monad is indexed [Atkey 2009] so as to keep track of the pre and post stack types of the generated bytecode.

Before showing how our writer monad is used to generate bytecode (§6), we present it in its general form. That is, we define it abstractly for a separation logic Pred A over any given PRSA A, and any given type $W : I \rightarrow I \rightarrow$ Pred A indexed by I that represents the generated output. The writer monad is then defined as follows:

$$\text{Writer} : (I \rightarrow I \rightarrow \text{Pred } A) \rightarrow I \rightarrow I \rightarrow \text{Pred } A \rightarrow \text{Pred } A$$
$$\text{Writer } W \ i_1 \ i_2 \ P \ = \ W \ i_1 \ i_2 \ * \ P$$

This definition resembles the usual definition of a writer monad. However, since the monad is defined in terms of separation logic, we use the separating conjunction $*$ instead of an ordinary product. The indices i_1 and i_2 of the monad are taken to be the indices of the output W.

To define the monadic operations of our writer monad, the type W for the output must be an *indexed monoid*. That means, we need to have operations mempty for the empty output, and mappend for appending output:

$$\text{mempty} \ : \ \epsilon[\ W \ i \ i \ ]$$
$$\text{mappend} \ : \ \forall[\ W \ i_1 \ i_2 \ \Rightarrow \ W \ i_2 \ i_3 \ \twoheadrightarrow \ W \ i_1 \ i_3 \ ]$$

These operations generalize the ordinary indexed monoid operations, but instead of using functions, we use the separation logic connectives.[4]

Apart from return and bind, our writer monad has two additional monadic operations: tell for generating output, and censor for transforming the output:

$$\text{return} \ : \ \forall[\ P \ \Rightarrow \ \text{Writer } W \ i_1 \ i_1 \ P \ ]$$
$$\text{bind} \quad : \ \forall[\ (P \ \twoheadrightarrow \ \text{Writer } W \ i_2 \ i_3 \ Q) \ \Rightarrow \ (\text{Writer } W \ i_1 \ i_2 \ P \ \twoheadrightarrow \ \text{Writer } W \ i_1 \ i_3 \ Q) \ ]$$
$$\text{tell} \quad : \ \forall[\ W \ i_1 \ i_2 \ \Rightarrow \ \text{Writer } W \ i_1 \ i_2 \ \text{Emp} \ ]$$
$$\text{censor} \ : \ \forall[\ (W \ i_1 \ i_2 \ \twoheadrightarrow \ W \ i_3 \ i_4) \ \Rightarrow \ \text{Writer } W \ i_1 \ i_2 \ P \ \twoheadrightarrow \ \text{Writer } W \ i_3 \ i_4 \ P \ ]$$

Similar to the monoid operations, the monadic operations generalize the ordinary (indexed) writer monad operations by using the separation logic connectives.

---

[4]There is some artistic freedom within these signatures. For mempty we could also have used $\forall[\ \text{Emp} \ \Rightarrow \ W \ i \ i \ ]$, and for mappend we could have used $\epsilon[\ W \ i_1 \ i_2 \ \twoheadrightarrow \ W \ i_2 \ i_3 \ \twoheadrightarrow \ W \ i_1 \ i_3 \ ]$. We recommend the reader to unfold the definition of the wands $\twoheadrightarrow$ to understand why these are semantically identical (and also, why $\epsilon[\ W \ i_1 \ i_2 \ \Rightarrow \ W \ i_2 \ i_3 \ \Rightarrow \ W \ i_1 \ i_3 \ ]$ is not). These equivalent variants are less convenient to be used for programming in Agda—at the call site they require additional trivial proof work. Hence, we consistently prefer the style where top level emptiness is captured using the ϵ-closure (instead of Emp $\Rightarrow$), and functions use $\Rightarrow$ as the top-level connective. However, when functions are used as first class values, one needs to use the wand $\twoheadrightarrow$.

Programming with monadic strength. While the monadic writer operations are elegant, the bind operation is hard to use in Agda. The bind operation is *internal*, and thus uses a wand $-\!*$ to represent the continuation. Unfortunately, constructing wands *in Agda* is unpleasant. The fact that a wand takes a separation witness as its argument, means that each time we use bind, we need to construct a continuation that takes a proof term of a separation witness that needs to be passed around. As a side effect, the internal bind cannot be used in combination with Agda's built-in **do**-notation.

To remedy this problem, we follow Poulsen et al. [2018] and Rouvoet et al. [2020a] by making use of the fact that the internal bind is equivalent in expressive power to the *external* bind $\_=\!\!\ll\!\_$ in combination with *monadic strength* [Kock 1972; Moggi 1991] over the separating conjunction $*$:

$$\_=\!\!\ll\!\_ \quad : \forall[\ P \Rightarrow \mathsf{Writer}\ W\ i_2\ i_3\ Q\ ] \ \rightarrow\ \forall[\ \mathsf{Writer}\ W\ i_1\ i_2\ P \Rightarrow \mathsf{Writer}\ W\ i_1\ i_3\ Q\ ]$$
$$\mathsf{strength} : \forall[\ \mathsf{Writer}\ W\ i_1\ i_2\ P \Rightarrow Q \ -\!*\ \mathsf{Writer}\ W\ i_1\ i_2\ (P * Q)\ ]$$

Using the external bind we can write our monadic computations using **do**-notation in Agda, which desugars in the usual way. We will show examples of this in §6.

4.4 Separation Logic for Interfaces

To specify intrinsically typed bytecode syntax (§5) and to implement our compiler (§6) we instantiate our separation logic from §4.2 with three different PRSAs: (1) disjoint context separation $\_\sqcup\_\simeq\_$ to abstract over exports (i.e., label binders), (2) context separation with overlap $\_\cup\_\simeq\_$ to abstract over imports (i.e., label references, and (3) interface composition $\_\bullet\_\simeq\_$ to abstract over interfaces. To convert between these three separation logic instances, we define modalities Export and Import for lifting objects from the logics of label contexts to the logic of interfaces:

data Export $(P : \mathsf{Pred\ LabelCtx}) : \mathsf{Pred\ Intf}$ **where**
   exports : $P\ E\ \rightarrow\ \mathsf{Export}\ P\ (E \Uparrow [])$

data Import $(P : \mathsf{Pred\ LabelCtx}) : \mathsf{Pred\ Intf}$ **where**
   imports : $P\ I\ \rightarrow\ \mathsf{Import}\ P\ ([] \Uparrow I)$

Both modalities are monotone and commute with separating conjunction $*$:

mapExport : $\forall[\ P \Rightarrow Q\ ]\ \rightarrow\ \forall[\ \mathsf{Export}\ P \Rightarrow \mathsf{Export}\ Q\ ]$
mapImport : $\forall[\ P \Rightarrow Q\ ]\ \rightarrow\ \forall[\ \mathsf{Import}\ P \Rightarrow \mathsf{Import}\ Q\ ]$
zipExport   : $\forall[\ (\mathsf{Export}\ P) * (\mathsf{Export}\ Q) \Rightarrow \mathsf{Export}\ (P * Q)\ ]$
zipImport   : $\forall[\ (\mathsf{Import}\ P) * (\mathsf{Import}\ Q) \Rightarrow \mathsf{Import}\ (P * Q)\ ]$

Note that the separating conjunction $*$ in Export $(P * Q)$ is using disjoint context separation $\_\sqcup\_\simeq\_$, whereas the $*$ in Import $(P * Q)$ is using context separation with overlap $\_\cup\_\simeq\_$, and the $*$ at the top-level is using interface composition $\_\bullet\_\simeq\_$. The fact that these modalities convert from one logic to another means that they are *relative* [Altenkirch et al. 2015].

The intuition for these dual modalities is that they generalize the dual roles of labels as either imports (i.e., binders) or exports (i.e., references). We can thus recover the interface predicates Binder and Reference from §2.4 as follows:

Binder ψ = Export $(\mathsf{Own}\ [\ \psi\ ])$
Reference ψ = Import $(\mathsf{Own}\ [\ \psi\ ])$

From now on, we will use the right-hand side of these definitions directly, highlighting the role of the modalities in the bytecode syntax and the monadic operations.

In §3.1 we showed how the exports and imports of interfaces interact. This interaction is internalized in separation logic over interfaces by the following operation:

freshLabels : ϵ[Export (Own E) ∗ Import (Own E)]

This operation generalizes the freshLabel operation we have seen in §2.4 from Binder and Reference to the Export and Import modalities, possibly returning multiple labels. Like matter and anti-matter, the operation freshLabels shows that labels can spontaneously come into existence in pairs of binding and reference occurrences. Binding occurrences are represented as ownership of a linear resource $E \upharpoonleft []$, whereas reference occurrences are represented as ownership of a relevant (i.e., duplicable) resources $[] \upharpoonleft E$. The definition of freshLabels relies on the fact that these resources are dual, and thus cancel each other out, i.e., $(E \upharpoonleft []) \bullet ([] \upharpoonleft E) \simeq \epsilon$.

Note that neither Export (Own E) nor Import (Own E) is affine/can be dropped. That is, we do not have ∀[Export (Own E) ⟹ Emp] nor ∀[Import (Own E) ⟹ Emp]. Formally, this corresponds to the fact that both imports and exports are treated relevantly in the co-contextual model of interface composition. Intuitively, this is explained by the fact that if we intend to make use of either Export (Own E) or Import (Own E), we are obliged to also use their dual counterpart. This ensures that references are well bound and there are no dangling binders.

5 INTRINSICALLY-TYPED NAMELESS CO-CONTEXTUAL BYTECODE

Using the separation logic as a logical framework, we now define intrinsically typed bytecode. We adopt the perspective that well-typed terms are separation logic predicates on interfaces, thus abstracting over co-contexts and their compositions.

Our typed bytecode language is a subset of JVM bytecode [Lindholm et al. 2020], but uses labels to denote jump targets. Labels can be translated away later in the compiler pipeline, after the bytecode is optimized (§7). Compared to the simple bytecode language from §2, our JVM subset additionally has local variables, and permits attaching multiple labels to a single instruction. The latter simplifies compilation, but does not change the verification problem that we described in §2.

The intrinsically typed bytecode syntax in Fig. 5 internalizes the co-contextual typing judgments shown in Fig. 3. We first define plain instructions Instr Γ ψ_1 ψ_2. Variable binding is modeled contextually using a local variable context Γ. As is usual, variables are modeled using proof-relevant membership of the context $a \in \Gamma$, pointing out a *specific* element in Γ. The indices ψ_1 and ψ_2 again denote the pre and post stack typing. The values of these indices for the various instructions are identical to the stack types in the extrinsic co-contextual rules. As before, plain instructions only import labels (i.e., only contain label references), and are thus typed as predicates over a LabelCtx. The jump instructions (**goto** and **iftrue**) are the only instructions that have imports, and use the predicate Own [ψ] to express that.

In contrast to instructions, code points Code Γ ψ_1 ψ_2 are predicates on interfaces Intf, because they import and export labels (i.e., contain label references and binders). The constructors **labeled** and **plain** use the Import modality to lift plain instructions into the logic of predicates on interfaces. The constructor **labeled** also attaches one or more binders to an instruction using the Export modality. For this it uses the type Labeling ψ, which represents one or more labels of the same type ψ, and is defined using a version of the Kleene plus operator Plus in separation logic.

Finally, bytecode Bytecode Γ ψ_1 ψ_2 is represented as the indexed reflexive-transitive closure (or Kleene star) IStar of code points. This makes bytecode an indexed monoid, that has the operations mempty and mappend that we described in §4.3 and are needed to use bytecode as the output of computations in the Writer monad in §6.2.

It is noteworthy that the data structures Star, Plus and IStar are defined generically over arbitrary PRSAs. This is a strength of the approach we have taken. By casting the invariants of label binding in the framework of proof-relevant separation logic, we have gained reuse of common data structures and operations that we otherwise would have had to tailor to a specific domain.

Auxiliary data types: Kleene star and plus, and indexed Kleene star

data Star (P : Pred A) : Pred A **where**   **data** IStar (R : $I \rightarrow I \rightarrow$ Pred A)
  nil   : ϵ[Star P]   : ($I \rightarrow I \rightarrow$ Pred A) **where**
  cons : $\forall$[$P * $ Star $P \Rightarrow$ Star P]   **inil**   : ϵ[IStar $R\ i\ i$]
Plus P = $P * $ Star P **icons** : $\forall$[$R\ i_1\ i_2 * $ IStar $R\ i_2\ i_3 \Rightarrow$ IStar $R\ i_1\ i_3$]

Types

data Ty : Set **where**   VarCtx   = List Ty
  boolean   : Ty   StackTy   = List Ty
  byte short int long char : Ty   LabelCtx = List StackTy

Intrinsically typed bytecode syntax

data Const : Ty $\rightarrow$ Set **where** (...)

data Instr (Γ : VarCtx) : ($\psi_1\ \psi_2$: StackTy) $\rightarrow$ Pred LabelCtx **where** (...)
  nop   : ϵ[Instr $\Gamma\ \psi\ \psi$]
  pop   : ϵ[Instr $\Gamma\ (a :: \psi)\ \psi$]
  push   : Const $a \rightarrow \epsilon$[Instr $\Gamma\ \psi\ (a :: \psi)$]
  swap   : ϵ[Instr $\Gamma\ (a :: b :: \psi)\ (b :: a :: \psi)$]
  iadd   : ϵ[Instr $\Gamma\ (\mathbf{int} :: \mathbf{int} :: \psi)\ (\mathbf{int} :: \psi)$]
  load   : $a \in \Gamma$ $\rightarrow \epsilon$[Instr $\Gamma\ \psi\ (a :: \psi)$]
  store   : $a \in \Gamma$ $\rightarrow \epsilon$[Instr $\Gamma\ (a :: \psi)\ \psi$]
  goto   : $\forall$[Own [ψ_1] $\Rightarrow$ Instr $\Gamma\ \psi_1\ \psi_2$]
  iftrue   : $\forall$[Own [ψ] $\Rightarrow$ Instr $\Gamma\ (\mathbf{boolean} :: \psi)\ \psi$]
Labeling ψ = Plus (Own [ψ])

data Code (Γ : VarCtx) : ($\psi_1\ \psi_2$: StackTy) $\rightarrow$ Pred Intf **where**
  labeled : $\forall$[Export (Labeling ψ_1) $*$ Import (Instr $\Gamma\ \psi_1\ \psi_2$) $\Rightarrow$ Code $\Gamma\ \psi_1\ \psi_2$]
  plain   : $\forall$[Import (Instr $\Gamma\ \psi_1\ \psi_2$) $\Rightarrow$ Code $\Gamma\ \psi_1\ \psi_2$]
Bytecode $\Gamma\ \psi_1\ \psi_2$ = IStar (Code Γ) $\psi_1\ \psi_2$

Fig. 5. Intrinsically typed nameless co-contextual bytecode.

6 COMPILING WITH LABELS

Having defined our bytecode language in the previous section, we now describe an intrinsically typed compiler that translates programs in an imperative source language with structured control into bytecode. We first define the well-typed syntax of the source language IMP (§6.1). We then define a monad Compiler, which extends the monad Writer with tailor-made operations for generating bytecode (§6.2), and is used to present our compiler (§6.3).

6.1 The Intrinsically Typed Source Language IMP

Fig. 6 shows the intrinsically typed syntax of the imperative language IMP [Nipkow and Klein 2014]. Like many imperative languages, it distinguishes expressions Exp Γ a and statements Stmt Γ (also known as *commands*). Expressions are pure and result in a value of type a, whereas statements are side-effecting but have no result. Expressions are constants (**num**, and **bool**), reads

```
data BinOp : (a b c : Ty) → Set where
  add sub mul div xor : BinOp int int int
  eq ne lt ge gt le   : BinOp int int bool
data Exp (Γ : VarCtx) : Ty → Set where
  num         : ℕ →              Exp Γ int
  bool        : Bool →           Exp Γ bool
  var         : a ∈ Γ →          Exp Γ a
  bop         : BinOp a b c →    Exp Γ a    → Exp Γ b → Exp Γ c
  if_then_else :                 Exp Γ bool → Exp Γ a → Exp Γ a → Exp Γ a
data Stmt (Γ : VarCtx) : Set where
  assign      : a ∈ Γ →          Exp Γ a         → Stmt Γ
  if_then_else :                 Exp Γ bool   → Stmt Γ → Stmt Γ → Stmt Γ
  while       :                  Exp Γ bool   → Stmt Γ → Stmt Γ
  block       :                  List (Stmt Γ) → Stmt Γ
```

Fig. 6. Intrinsically typed syntax of the imperative language IMP.

from local variables (var), binary operations for arithmetic and comparison (bop), or conditionals (if_then_else). Statements are assignments to local variables (assign), conditions (if_then_else), while loops (while), or multiple statements sequenced in a block (block).

The language IMP misses a statement for local variables declarations. The compiler backend that we present in §7 will start with an extended source language IMP⁺ that includes local variable declarations. The first pass translates IMP⁺ to IMP by hoisting local variables declarations. This separate pass simplifies the compilation to bytecode as it allows us to use the same context Γ for the source Exp Γ a and target Bytecode Γ ψ_1 ψ_2 of the compiler.[5]

In IMP⁺, local variable declarations come with initializers, which ensures that variables are initialized before use. This is, however, not witnessed by the typing of IMP, where the first assignment is separated from the hoisted declaration. Intrinsically capturing the effect analysis for variable initialization is an interesting direction for future work, orthogonal from the verification problem with labels that we address in this paper.

6.2 The Compiler Monad

Like the untyped compiler in Fig. 1, we use a writer monad to keep track of the generated bytecode output. For this purpose we use the indexed Writer monad defined in §4:

$$\text{Compiler } \Gamma \ \psi_1 \ \psi_2 \ = \ \text{Writer (Bytecode } \Gamma) \ \psi_1 \ \psi_2$$

Note that Writer requires Bytecode Γ to be an indexed, resource-aware monoid. We have shown that this is the case in §5. The indices ψ_1 and ψ_2 of the Writer are used to type the generated bytecode output. To ease programming with Compiler, we define two derived monadic operations. Using tell, we define the operation code, which sends a single unlabeled instruction to the output. Using censor, we define the operation attach for labeling the first instruction of the generated output with the label ℓ passed as an argument:

---

[5]For simplicity we use the same set of types for the IMP and bytecode language. The artifact [Rouvoet et al. 2020b] uses different sets of types for both languages. It shows how one can translate almost transparently between them using Agda's mechanism for type class search and rewriting [Cockx 2020].

```
oneplus    : ∀[ P  ⇒  Plus P ]
oneplus p  =  p •⟨ •-idʳ ⟩ nil

oneistar   : ∀[ R i₁ i₂  ⇒  IStar R i₁ i₂ ]
oneistar r  =  icons (r •⟨ •-idʳ ⟩ inil)

code       : ∀[ Import (Instr Γ ψ₁ ψ₂)  ⇒  Compiler Γ ψ₁ ψ₂ Emp ]
code i     =  tell (oneistar (plain i))

attach     : ∀[ Export (Own [ ψ₁ ])  ⇒  Compiler Γ ψ₁ ψ₂ P  —∗  Compiler Γ ψ₁ ψ₂ P ]
attach ℓ   =
   censor (mappend (oneistar (labeled ((oneplus ⟨$⟩ ℓ) •⟨ •-idʳ ⟩ (imports nop)))))
```

The first argument of censor of type Bytecode Γ ψ_1 ψ_2 —∗ Bytecode Γ ψ_1 ψ_2 is a function that transforms the output of the compiler computation and can own resources. We pass it the function mappend (..) that prepends a singleton bytecode sequence, consisting of a nop instruction labeled with the label passed to attach. To convert the label ℓ to a Labeling, we use the functorial map ⟨$⟩ on Export to lift the function oneplus that constructs a singleton labeling.

The definition of attach shows that label binders (and references for that matter) can be treated as first-class values using the linear typing discipline. It is useful to consider what happens under the surface here. Recall that binders are represented in an nameless fashion so that references to them are only given meaning by the relationship depicted by the interface composition witnesses. Hence, attaching the binder ℓ is only a meaningful operation if the interface compositions that occur in the output of attach, and between its output and the surrounding bytecode, are exactly the right proof terms. Yet, thanks to the abstraction of the logic, the programmer is not bothered with any of this. Instead, they are only concerned with satisfying the resource constraints that the type-checker enforces, which in this case means that they can only use ℓ once.

6.3 The Compiler From IMP to Bytecode

We now put all ingredients we developed together to define an intrinsically typed compiler from IMP to Bytecode. We focus on the compilation of expressions, which is done using:

```
compileₑ  : Exp Γ a  →  ϵ[ Compiler Γ ψ (a :: ψ) Emp ]
```

This signature expresses concisely the stack invariant of expressions—it says that the bytecode will operate in any stack ψ, and leave behind a single value whose type a matches the expression's type. The compiler is defined by pattern matching on the expression. As a warm-up exercise, let us consider the compilation of constants, which translate to a single push instruction:

```
compileₑ (num x)  =  do                    compileₑ (var x)  =  do
   code (imports (push (num x)))              code (imports (load x))
```

The integration of the stack invariant in the type of the compiler, prevents us from writing bytecode that is not stack safe. Agda's type checker ensures that we meet the stack postcondition, and do not make any unwarranted assumptions about the input stack ψ. For example:

```
compileₑ (num x)  =  do                    compileₑ (bool b)  =  do
   code (imports (push (bool true)))          code (imports swap)
   ⊘ error: boolean ≠ int                     ⊘ error: _a_82 :: _b_83 :: _ψ_84 ≠ ψ
```

The compilation of constructs that involve control flow is a little more involved as we have to generate labels. The compilation of conditionals if c then e_1 else e_2 is as follows:

```
compileₑ (if c then e₁ else e₂) = do
  compileₑ c
  let (ℓ⁻_then •⟨ σ₁ ⟩ ℓ⁺_then)       =  *-swap freshLabels
  ℓ⁺_then                             ←  *−id⁻ˡ ⟨$⟩ (code (iftrue ⟨$⟩ ℓ⁻_then) &⟨ σ₁ ⟩ ℓ⁺_then)
  compileₑ e₂
  let (ℓ⁻_end •⟨ σ₂ ⟩ labels)         =  *-rotateₗ (*-assocᵣ (freshLabels •⟨ •-idʳ ⟩ ℓ⁺_then))
  ℓ⁺_then •⟨ σ₃ ⟩ ℓ⁺_end              ←  *−id⁻ˡ ⟨$⟩ (code (goto ⟨$⟩ ℓ⁻_end) &⟨ σ₂ ⟩ labels)
  ℓ⁺_end                              ←  *−id⁻ˡ ⟨$⟩ (attach ℓ⁺_then •-idʳ (compileₑ e₁) &⟨ σ₃ ⟩ ℓ⁺_end)
  attach ℓ⁺_end •-idʳ (return refl)
```

This code consists of the same eight operations as used in the untyped compiler in Fig. 1. The most significant difference in appearance arises due to the need to use monadic strength $mp \,\&\langle\, \sigma \,\rangle\, q$ (which denotes strength $mp \,\langle\, \sigma \,\rangle\, q$, and binds very loosely) to carry values across operations. Here, σ witnesses separation between mp and q. Using strength, we build up a context of values that own resources, represented as a big separating conjunction. We use the laws of the separation logic (e.g., $*-\mathsf{id}^{-l}$ and $*\text{-rotate}_l$) to simplify and reorder this context so that values appear in the right order. When using values, we have to provide the separation witness that relates the resource that they own to the resources owned by the context. This witness comes from the deconstruction of the monadic context on the left, or is trivial if the monadic operation is resource neutral.

As an example, we consider the last three lines of the compilation. The first of these returns a context of two binding occurrences of labels ℓ^+_{then} and ℓ^+_{end}, separated by (the interface composition) σ_3. We attach ℓ^+_{then}, and keep ℓ^+_{end} in the context, to be attached to the end of the output. We do this with monadic strength, using σ_3 as evidence that label to be attached is indeed separated from the context. In the last line we finally attach ℓ^+_{end}, which leaves the context empty, and the resources balanced. If we would replace the last line with return refl, and thus create a dangling reference, Agda would reject the definition, because the resources that ℓ^+_{end} owns were dropped.

Compilation of the other expression constructs of IMP follows a similar pattern. The same is true for the compilation of statements, with the only significant difference being the type of the compiler, which obeys by a different stack invariant, leaving the stack exactly as it found it.

Beyond programming with strength. To use Agda's pattern matching and do-notation, we resort to programming with the external bind and monadic strength, rather than programming entirely within the logic using the internal bind. It would be interesting to extend the syntax of the host language (i.e., Agda or Idris) to make the latter practical [Brady and Hammond 2012].

7 AN INTRINSICALLY TYPED COMPILER BACKEND IN AGDA

Our intrinsically typed compiler from IMP to bytecode (§6) is part of a typed compiler backend pipeline that can be summarized as follows:
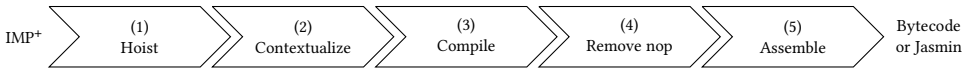
The compilation starts with typed IMP⁺ where lexical variable binding is represented co-contextually using the co-de-Bruijn encoding [McBride 2018]. The co-contextual typing is useful (1), because it allows hoisting declarations without weakening typed IMP⁺ code to account for the additional binding. The hoisting itself is implemented in a monad that collects the binders. After hoisting, there are no more local variable declarations left, giving us co-contextually typed IMP. The *contextualize* transformation (2) then eliminates the co-de-Bruijn encoding to contextually typed IMP (whose syntax is in Fig. 6). A generic version of transformation (2) is given by McBride [2018].

As shown in §6, we then compile IMP to co-contextually typed bytecode. The compiler inserts some unnecessary nop instructions. We remove those unnecessary instructions using a simple intrinsically typed bytecode optimization pass. This transformation is pure, and implemented in 17 lines of Agda. Like compilation it requires no lemmas about bytecode except the axioms of proof-relevant separation algebras. Bytecode optimizations are simplified greatly by representation that use labels, because unlike absolute or relative addresses, labels can be moved.

The final stage of our compiler backend is eliminating labels altogether, replacing them by absolute addresses of instructions. We implement this transformation in an intrinsically typed style. The key ideas of the transformation were explained in §3.3. Alternatively, one can use our implementation of an unverified transformation to Jasmin code, which can be assembled to a Java class file and run by the JVM.

Our compiler pipeline lacks a frontend for parsing and type-checking IMP⁺ programs. To test our compiler pipeline we have manually embedded a couple of IMP⁺ programs in Agda, and used Agda's extraction to Haskell to obtain the generated bytecode.

The code that we presented in this paper can be found in the accompanying artifact [Rouvoet et al. 2020b][6]. Most of the code passes the Agda type checker under the --safe flag. Exceptions to this are the hoisting, contextualization, and assembly transformations. The former two do not pass the termination checker as implemented, and the latter uses Agda's rewriting mechanism [Cockx 2020]. The compiler pipeline and the source, target, and intermediate languages are implemented in ~1700 lines of Agda. The library for separation logic that we used and contributed to, consists of ~4500 lines. This includes many instances of PRSAs, as well as many generic data structures and monads. The PRSA transformer version of the model described in §3 has been added to the library.

8 RELATED WORK

Co-contextual and principal typing. We faced the problem that the invariant of compilation to a low-level language with label binding was not preserved by the individual compiler operations. We addressed this problem by reformulating bytecode typing co-contextually. The co-contextual typings are *more principal* [Jim 1996] than contextual typings, because the encapsulated knowledge about labels is restricted to be the minimal amount required to type the bytecode fragment. That is, they are a principal representative for many typings that can conceptually be obtained by weakening the context.[7]

From this perspective, we can think of our nameless and co-contextual encoding of label binding as a way to push the limits of what properties of our bytecode languages are principal. By going nameless, we were able to decide at the *outside* of a bytecode fragment how labels defined *inside* of it relate to other labels. In other words, the nameless encoding is a *principal labeling*, where we assume less about labels that are used, and can use labels that are defined in more ways.

A key point of these principal typings is the *compositional* nature, which accommodates separate analysis, but can also help to define *total* functions on typed terms [Wells 2002]. Examples of work that exploit this, knowingly or unknowingly, are plentiful. For example, there are lines of work on incremental type checking [Erdweg et al. 2015; Kuci et al. 2017], and separate compilation [Ancona et al. 2004; Jim 1996]. Notably, the idea of treating exports and imports of names as opposites in order to give co-contextual accounts of global binding, is already present in the (not mechanically

---

[6]The artifact is also available as a virtual machine [Rouvoet et al. 2020c].

[7]Our bytecode typing is not entirely principal. In particular, for every concrete bytecode fragment we assume that we know enough of the context to type the entire stack. This proved sufficient for compilation, where we always have enough knowledge about the stack. We also compile to JVM bytecode, which requires that the entire stack can be typed with a *monomorphic* type. This is different for some other typed low-level languages, such as typed assembly language [Morrisett et al. 1999a,b], which support more principal typings for stacks via stack polymorphism.

verified) work on co-contextual type checking by Kuci et al. [2017], where references to globally defined names yield constraints that are removed at the top-level by corresponding declarations.

McBride [2018] proposed the co-de-Bruijn encoding of lexical binding in a lambda calculus so that hereditary substitution on well-typed terms becomes a total function. Inspired by that encoding, Rouvoet et al. [2020a] give a typing of linear references in a session-typed language, incorporating the two roles of supply and demand in a state monad. We are inspired by both these works in the construction of our nameless model for global label binding. In retrospect, both are co-contextual reformulations of existing type-systems, resulting in more compositional typings.

The flip side of these more principal typings is that work is moved from the leaves of term typing to the nodes. This is visible in all the work cited above, and is also present in our work in the premises about interface composition. McBride [2018] already introduces *relevant pairs* as the right notion of composition. Rouvoet et al. [2020a] identify this notion as the separating conjunction and extend it to a complete proof-relevant separation logic. In this paper, we provide a new example of the applicability of Rouvoet et al.'s logic, provide new data structures, and computational structures built on top of this logic, and also construct the modalities Export and Import that are specific to the instantiation of separation logic with interfaces.

Having identified this relation between co-contextual/principal typings of terms and proof-relevant separation logic, it would be interesting to apply this approach to other invariants that appear anti-compositional, and are thus hard to express intrinsically. Both to systematize the approach, and to design a better dependently typed meta-language that can further simplify the definition of functions that utilize the framework, like our compiler.

Intrinsic verification of compilers. Abel [2020] presented a well-typed compiler that targets a typed representation of control-flow graphs. In his intermediate language, control-flow is reduced to two primitive constructs: *join* points and *looping* points. Unlike bytecode this is not a flat language, which allows labels to be treated like lexical variables in e.g., the well-typed syntax of STLC. A second compiler pass (linearization) must then translate control flow graphs to a flat bytecode language, but their implementation of this pass has not been published. The author does note that the full verification requires reasoning in the sublist category. We have shown that we can avoid such reasoning by working with nameless binding in an embedded linear language.

McBride [2012] showed how to intrinsically verify *functional* correctness of a compiler for a small arithmetic expression language by indexing syntax by their semantics. This is extended by Pardo et al. [2018] with top-level variables, and sequenced and looping assignments to those variables. They also compile via an intermediate language with sequencing and loops, and output low-level bytecode with relative jumps. Although the intermediate language is indexed by a semantics, the low-level target language is not. Instead they give this semantics extrinsically, because it is unclear whether it is possible to give a "syntax-directed" semantics for the low-level language with jumps.

Extrinsic compiler verification. There is a lot of prior work on verifying correctness results of compilers and compiler optimizations in proof assistants. Compilers often compile via various intermediate languages to simplify transformations and their verification [Kumar et al. 2014; Leroy 2009]. *Control-flow graphs* and *continuation-passing style* are commonly used intermediate representations in compilers (see for example Appel [2006]; Bélanger et al. [2015]; Chlipala [2007]; Guillemette and Monnier [2008]; Morrisett et al. [1999b]). Such intermediate languages are a more suitable level of abstraction for expressing high-level compiler optimizations. After applying the optimizing transformations, these higher-level representations of control-flow are eliminated in favor of jumps and labels in a low-level machine language. In a well-typed version of such a multi-stage compiler, our compilation monad would only be used for this pass, which is usually called *linearization.* The problem that verifying label binding requires additional reasoning in the compiler

definition, is not an issue in these works, because all the verification is done extrinsically. Good extrinsic proof automation can be an alternative approach to avoiding manual proof overhead.

In extrinsic verification, two other methods have been used to denote jump targets: instruction addresses (e.g., in the machine language of CakeML [Kumar et al. 2014]) and instruction offsets (e.g., in the Jinja compiler [Klein and Nipkow 2006]). Although both are candidates for intrinsically typed bytecode representations, they cannot be used in the compilations that we have used throughout this paper that output forward jumps before recursively compiling the instructions that are targeted. Both encodings require computing the output up-to the jump target first, so as to know the address of the target. In addition, it is difficult to write bytecode transformations using absolute or relative addresses. Even our nop removal transformation would require shifting jumps throughout the program. In label-based representations this is much simpler, because labels can be moved.

The design of more expressive type systems for assembly languages [Crary 2003; Morrisett et al. 1999b], is an important, but largely orthogonal research direction. The same is true for research into semantic type systems for machine languages, and verification techniques that reduce the trusted computing base [Appel 2001].

Linear meta-languages. We have constructed a shallowly embedded language that tracks resource usage. Another approach is to use a meta language with built-in support for resources, like linear Haskell [Bernardy et al. 2018], Idris [Brady 2013] (where Idris 2 implements quantitative type theory (QTT) [Atkey 2018]), or Granule [Orchard et al. 2019]. To be able to use such a language instead of our separation logic, the language needs to support not only user-defined resources, but also *proof-relevant* resources. Our logic not only hides the accounting at the time of type-checking the compiler, but also the construction of a proof term that exists at runtime. We implement the primitives (freshLabels) using the fact that the logic is a shallow embedding. When we eliminate labels in the assembler, we again poke through the abstractions of the logic, and compute directly on the proof terms. To the best of our knowledge there is no language with support for a resource such as our interfaces. Granule appears to come the nearest to this, as its theory permits user-defined resources. The current implementation, however, does not. A resource in Granule and QTT must be a semiring with functional operations [Orchard et al. 2019], and thus does not support PRSAs. Despite the fact that an expressive enough language does not yet exist, this is a promising path towards a more suitable dependently typed meta language.

9 CONCLUSION

We presented the intrinsically typed compilation of a small language with structured control flow to typed bytecode, giving rise to a compiler that is type correct by construction. The key problem we faced was the fact that label well-boundedness appeared to be an anti-compositional, whole-program property. Our first key idea to solve this problem is to reformulate label well-boundedness using a nameless and co-contextual representation, which turns it into a compositional, local property. Our second key idea is to abstract over co-contexts and their compositions using a proof-relevant separation logic. As a result, our intrinsically typed compiler, as well as the operations that support it, contain little manual proof work and mirror their untyped counterparts.

REFERENCES

Andreas Abel. 2020. Type-preserving compilation via dependently typed syntax in Agda. Abstract of a talk at TYPES, available online at http://www.cse.chalmers.se/~abela/types20.pdf, slides available online at http://www.cse.chalmers.se/~abela/talkTYPES2020.pdf.

Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. 2015. Monads need not be endofunctors. *Logical Methods in Computer Science (LMCS)* 11, 1 (2015). https://doi.org/10.2168/LMCS-11(1:3)2015

Davide Ancona, Ferruccio Damiani, Sophia Drossopoulou, Elena Zucca, et al. 2004. Even more principal typings for Java-like languages. In *Formal Techniques for Java-like Programs Workshop (FTfJP)*.

Andrew W. Appel. 2001. Foundational proof-carrying code. In *Logic in Computer Science (LICS)*. 247–256. https://doi.org/10.1109/LICS.2001.932501

Andrew W. Appel. 2006. *Compiling with continuations*. Cambridge University Press. https://doi.org/10.1017/CBO9780511609619

Robert Atkey. 2009. Parameterised notions of computation. *Journal of Functional Programming (JFP)* 19, 3–4 (2009), 335–376. https://doi.org/10.1017/S095679680900728X

Robert Atkey. 2018. Syntax and semantics of quantitative type theory. In *Logic in Computer Science (LICS)*. 56–65. https://doi.org/10.1145/3209108.3209189

Lennart Augustsson and Magnus Carlsson. 1999. An exercise in dependent types: A well-typed interpreter. In *Workshop on Dependent Types in Programming*.

Olivier Savary Bélanger, Stefan Monnier, and Brigitte Pientka. 2015. Programming type-safe transformations using higher-order abstract syntax. *Journal of Formalized Reasoning (JFR)* 8, 1 (2015), 49–91. https://doi.org/10.6092/issn.1972-5787/5122

Nick Benton, Chung-Kil Hur, Andrew Kennedy, and Conor McBride. 2012. Strongly typed term representations in Coq. *Journal of Automated Reasoning (JAR)* 49, 2 (2012). https://doi.org/10.1007/s10817-011-9219-0

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2018. Linear Haskell: Practical linearity in a higher-order polymorphic language. *PACMPL* 2, ACM SIGPLAN Symposium on Principle of Programming Languages (POPL) (2018), 5:1–5:29. https://doi.org/10.1145/3158093

Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *The Journal of Functional Programming (JFP)* 23, 5 (2013), 552. https://doi.org/10.1017/S095679681300018X

Edwin C. Brady and Kevin Hammond. 2012. Resource-safe systems programming with embedded domain specific languages. In *Practical Aspects of Declarative Languages (PADL)*. 242–257. https://doi.org/10.1007/978-3-642-27694-1_18

Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. 2007. Local Action and Abstract Separation Logic. In *Logic in Computer Science (LICS)*. 366–378. https://doi.org/10.1109/LICS.2007.30

Adam Chlipala. 2007. A certified type-preserving compiler from lambda calculus to assembly language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 54–65. https://doi.org/10.1145/1250734.1250742

Jesper Cockx. 2020. Type Theory Unchained: Extending Agda with User-Defined Rewrite Rules. In *Types for Proofs and Programs (TYPES) (Leibniz International Proceedings in Informatics (LIPIcs))*, Vol. 175. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2:1–2:27. https://doi.org/10.4230/LIPIcs.TYPES.2019.2

Karl Crary. 2003. Toward a foundational typed assembly language. In *ACM SIGPLAN Symposium on Principle of Programming Languages (POPL)*. 198–212. https://doi.org/10.1145/640128.604149

Robert Dockins, Aquinas Hobor, and Andrew W. Appel. 2009. A fresh look at separation algebras and share accounting. In *Asian Symposium on Programming Languages and Systems (APLAS) (LNCS)*, Vol. 5904. 161–177. https://doi.org/10.1007/978-3-642-10672-9_13

Sebastian Erdweg, Oliver Bracevac, Edlira Kuci, Matthias Krebs, and Mira Mezini. 2015. A co-contextual formulation of type rules and its application to incremental type checking. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA)*. 880–897. https://doi.org/10.1145/2814270.2814277

Louis-Julien Guillemette and Stefan Monnier. 2008. A type-preserving compiler in Haskell. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 75–86. https://doi.org/10.1145/1411204.1411218

Trevor Jim. 1996. What are principal typings and what are they good for?. In *ACM SIGPLAN Symposium on Principle of Programming Languages (POPL)*. 42–53. https://doi.org/10.1145/237721.237728

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. *Journal of Functional Programming (JFP)* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Gerwin Klein and Tobias Nipkow. 2006. A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 28, 4 (2006), 619–695. https://doi.org/10.1145/1146811

Anders Kock. 1972. Strong functors and monoidal monads. *Archiv der Mathematik* 23, 1 (1972), 113–120.

Edlira Kuci, Sebastian Erdweg, Oliver Bracevac, Andi Bejleri, and Mira Mezini. 2017. A co-contextual type checker for Featherweight Java. In *European Conference on Object-Oriented Programming (ECOOP)*. 18:1–18:26. https://doi.org/10.4230/LIPIcs.ECOOP.2017.18

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A verified implementation of ML. In *ACM SIGPLAN Symposium on Principle of Programming Languages (POPL)*. 179–192. https://doi.org/10.1145/2535838.2535841

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, and Daniel Smith. 2020. The Java Virtual Machine specification: Java SE 14 edition. Available online at https://docs.oracle.com/javase/specs/jvms/se14/jvms14.pdf.

Conor McBride. 2012. Agda-curious? https://dl.acm.org/doi/pdf/10.1145/2364527.2364529 Keynote at the ACM SIGPLAN International Conference of Functional Programming (ICFP).

Conor McBride. 2018. Everybody's got to be somewhere. In *Workshop on Mathematically Structured Functional Programming (MSFP) (EPTCS)*, Vol. 275. 53–69. https://doi.org/10.4204/EPTCS.275.6

Conor Thomas McBride. 2014. How to keep your neighbours in order. In *ACM SIGPLAN International Conference of Functional Programming (ICFP)*. 297–309. https://doi.org/10.1145/2628136.2628163

James McKinna and Joel Wright. 2006. A type-correct, stack-safe, provably correct, expression compiler. Unpublished draft.

Robin Milner. 1978. A theory of type polymorphism in programming. *Journal of Computer and System Sciences (JCSS)* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

Eugenio Moggi. 1991. Notions of computation and monads. *Information and Computation* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. 1999a. TALx86: A realistic typed assembly language. In *Workshop on Compiler Support for System Software*. 25–35.

Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1999b. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21, 3 (1999), 527–568. https://doi.org/10.1145/319301.319345

Tobias Nipkow and Gerwin Klein. 2014. IMP: A Simple Imperative Language. In *Concrete Semantics*, Benjamin C Pierce (Ed.). Springer, Chapter 7, 75–94. https://doi.org/10.1007/978-3-319-10542-0_7

Ulf Norell. 2009. Dependently typed programming in Agda. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI)*. 1–2. https://doi.org/10.1145/1481861.1481862

Peter W. O'Hearn and David J. Pym. 1999. The logic of bunched implications. *Bulletin of Symbolic Logic* 5, 2 (1999), 215–244. https://doi.org/10.2307/421090

Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative program reasoning with graded modal types. *PACMPL* 3, ACM SIGPLAN International Conference on Functional Programming (ICFP) (2019), 110:1–110:30. https://doi.org/10.1145/3341714

Alberto Pardo, Emmanuel Gunther, Miguel Pagano, and Marcos Viera. 2018. An internalist approach to correct-by-construction compilers. In *International Symposium on Principles and Practice of Declarative Programming (PPDP)*. 17:1–17:12. https://doi.org/10.1145/3236950.3236965

Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. 2018. Intrinsically-typed definitional interpreters for imperative languages. *PACMPL* 2, ACM SIGPLAN Symposium on Principle of Programming Languages (POPL) (2018), 16:1–16:34. https://doi.org/10.1145/3158104

John C Reynolds. 2000. The meaning of types from intrinsic to extrinsic semantics. *BRICS Report Series* 7, 32 (2000).

Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. 2020a. Intrinsically-typed definitional interpreters for linear, session-typed languages. In *ACM SIGPLAN Conference on Certified Programming and Proofs (CPP)*. 284–298. https://doi.org/10.1145/3372885.3373818

Arjen Rouvoet, Robbert Krebbers, and Eelco Visser. 2020b. Intrinsically Typed Compilation with Nameless labels: Agda Sources. https://doi.org/10.5281/zenodo.4072068

Arjen Rouvoet, Robbert Krebbers, and Eelco Visser. 2020c. Intrinsically Typed Compilation with Nameless labels: Virtual Machine. https://doi.org/10.5281/zenodo.4071954

David Walker. 2005. Substructural type systems. In *Advanced topics in types and programming languages*, Benjamin C Pierce (Ed.). The MIT press, Chapter 1, 3–43.

Joe B. Wells. 2002. The essence of principal typings. In *International Colloquium on Automata, Languages, and Programming (ICALP)*. 913–925. https://doi.org/10.1007/3-540-45465-9_78