Safety First: How to Safely Disregard Unsafe Behaviour in Compiler Calculations

PATRICK BAHR, IT University of Copenhagen, Denmark

Compiler calculation is a technique for deriving a correct-by-construction compiler from the specification of the compiler's correctness. In this setting, the compiler specification typically states that the semantics of each compiled program is *bisimilar* to the semantics of the original source program, i.e. both programs have the same behaviour. However, full bisimilarity for all source programs is too strong a requirement for complex source languages with unsafe behaviour for which the compiler need not make any guarantees, e.g. because programs that exhibit unsafe behaviour are ruled out by the type checker. This has long been recognised and exploited in compiler verification, but to date no calculation technique can handle such partial specifications. To address this, we propose a generalisation of bisimilarity, called *skew bisimilarity*, that allows us to weaken the compiler specification so that we may safely ignore unsafe behaviour when calculating a compiler for the specification. We demonstrate that skew bisimilarity enables us to derive compilers that produce more efficient code compared to previous compiler calculation techniques, all while maintaining the same strong correctness guarantees for safe source programs.

We further show that – even for source languages without unsafe behaviour – skew bisimilarity provides a powerful generalisation of bisimilarity that enables a novel calculation technique for reasoning about *register machines*. This improves on existing compiler calculation techniques for register machines, which are currently limited to terminating source languages without effects. To demonstrate the effectiveness of skew bisimilarity as a proof technique for compiler calculation, we have fully formalised it in Agda and used this formalisation to calculate compilers for a variety of languages, including the first calculation of a compiler for a typed concurrent lambda calculus that targets a register machine.

CCS Concepts: • **Software and its engineering → Compilers**; **Formal software verification**; • **Theory of computation → Logic and verification**; **Program verification**.

Additional Key Words and Phrases: program calculation, bisimilarity, choice trees, register machine

ACM Reference Format:
Patrick Bahr. 2026. Safety First: How to Safely Disregard Unsafe Behaviour in Compiler Calculations. *Proc. ACM Program. Lang.* 10, ICFP, Article 285 (August 2026), 34 pages. https://doi.org/10.1145/3828683

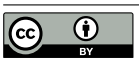
1 Introduction

Program calculation is a technique to derive correct-by-construction programs from specifications of their desired behaviour [Backhouse 2003; Bird 1989a,b; Morgan 1994]. Applied to compilers, calculation allows us to *discover* new compilation techniques along with proofs of their correctness. Even the instruction set targeted by the compiler may be discovered using calculation [Meijer 1992]. In its simplest form, the specification of a compiler is an equation stating that the semantics $[\![e]\!]$ of any source program e must be the same as the semantics of the compiled program $\mathsf{compile}(e)$:

$$[\![e]\!] \cong [\![\mathsf{compile}(e)]\!]$$

The *bisimilarity* relation $\cong$ demands that both sides exhibit the same observable behaviour.

Author's Contact Information: Patrick Bahr, IT University of Copenhagen, Copenhagen, Denmark, paba@itu.dk.



ACM 2475-1421/2026/8-ART285
https://doi.org/10.1145/3828683

However, in practice we do not expect compilers to satisfy the above equation for *all* programs, but only for those that are *safe*. Specifications that discount unsafe behaviours of source programs afford the compiler more flexibility to produce more efficient code. This is the typical approach taken by verified compilers such as CompCert [Leroy 2009] or CakeML [Kumar et al. 2014].

In this article, we propose a weaker form of bisimilarity, called *skew bisimilarity* and denoted by $\mathrel{{}_{\perp}\!\cong}$, that allows us to formulate a *partial* compiler specification in the form of an *inequality*

$$[\![e]\!] \mathrel{{}_{\perp}\!\cong} [\![\mathsf{compile}(e)]\!]$$

This specification makes no demands on unsafe behaviours exhibited by the left-hand side. While not an equivalence relation, skew bisimilarity still enjoys the equational reasoning principles that are necessary for calculating a correct-by-construction compiler. Moreover, any compiler that satisfies such a partial specification also satisfies the full correctness property for all safe programs:

$$[\![e]\!] \cong [\![\mathsf{compile}(e)]\!] \qquad \text{whenever } e \text{ is safe}$$

Skew bisimilarity both simplifies the equational reasoning for compiler calculations involving unsafe languages and can produce compilers that generate more efficient code compared to calculations that use strict bisimilarity.

We also show that skew bisimilarity has applications beyond source languages with unsafe behaviours. Even in cases where all programs of the source language are safe, the use of a partial specification enables a new compiler calculation technique to derive compilers that target *register machines* instead of stack machines. While the source language may be without unsafe behaviours, reading from registers may in general be unsafe, and skew bisimilarity allows us to elegantly reason in the presence of such unsafe behaviours. Importantly, even though the specification is partial and thus allows us to interact with unsafe register access, we nonetheless obtain the full correctness property in terms of strict bisimilarity.

We demonstrate this new compiler calculation technique enabled by skew bisimilarity on a variety of examples. For the purpose of exposition, we focus on simple toy languages in this article, but the accompanying Agda formalisation [Bahr 2026] calculates compilers for more feature-rich languages, including lambda calculi with features such as algebraic effects and concurrency. We stress that the purpose of compiler calculation is not to synthesise a full compiler implementation, which requires more than just code generation, but rather to discover compilation techniques for new languages in a rigorous and systematic manner. To demonstrate this, we have calculated a compiler for a higher-order functional reactive programming language with modal types (cf. section 4.4.1).

The rest of the article proceeds as follows: In section 2, we calculate a compiler for a simple language with unsafe behaviour. The example language is simple enough so that we can use the *Maybe* monad as the underlying semantic structure, but for more realistic languages we need the richer semantic structure provided by choice trees [Chappe et al. 2025], which we introduce in section 3. In section 4, we then introduce the skew bisimilarity relation on choice trees and apply it to a non-deterministic language with side effects. In section 5, we show how a mild generalisation of skew bisimilarity allows us to calculate compilers that target register machines instead of stack machines. Finally, we give an overview of related work in section 6 and conclude in section 7.

To make this article more accessible, we use Haskell notation as our metalanguage for all programs and calculations, but we assume that the metalanguage is total. All definitions, theorems, reasoning principles, and compiler calculations presented in this article have been formalised in Agda [Norell 2007] and are available as supplementary material [Bahr 2026].

2 Partial Semantics

In this section, we show how to calculate a compiler in the presence of unsafe behaviours in the source language semantics. For the sake of exposition, we choose a simple arithmetic expression language extended with Booleans and a conditional operator:

```
data Value = B Bool | N Int
data Expr  = Val Value | Add Expr Expr | If Expr Expr Expr
```

Since this expression language can manipulate two different types of values – integers and Booleans – untyped expressions may exhibit unsafe behaviour. However, the semantics is simple enough so that it can be formalised and reasoned about very easily. We start by defining the semantics of the expression language by a simple interpreter *eval* that evaluates expressions to values:

```
eval :: Expr → Value
eval (Val v)     = v
eval (Add x y) = let N m = eval x; N n = eval y in N (m + n)
eval (If x y z) = let B b = eval x in if b then eval y else eval z
```

Addition expects its arguments to evaluate to a number, whereas the conditional expects a Boolean as its first argument. The definition uses non-exhaustive pattern matching and thus evaluation fails on ill-typed expressions. For example, *eval* (*Add* (*Val* (*B True*)) (*Val* (*N* 9))) is not defined. The above definition of *eval* would only work if the metalanguage were not total. Instead of relying on a non-total metalanguage, we want to make partiality explicit so that we can reason about it. Since the expression language is terminating, the *Maybe* monad suffices to model partiality:

```
data Maybe a = Just a | Nothing

return :: a → Maybe a
return = Just

(≫=) :: Maybe a → (a → Maybe b) → Maybe b
Nothing ≫= f = Nothing
Just x  ≫= f = f x
```

Using *Maybe*, we can revise *eval* so that a type mismatch produces the result value *Nothing*:

```
eval :: Expr → Maybe Value
eval (Val v)     = return v
eval (Add x y) = do N m ← eval x; N n ← eval y; return (N (m + n))
eval (If x y z) = do B b  ← eval x; if b then eval y else eval z
```

To reduce syntactic clutter, we use Haskell's **do** notation, which translates into applications of ≫=. In addition, we use the fact that *Maybe* is an instance of the *MonadFail* type class, which provides a *fail* method that is invoked whenever a non-exhaustive pattern match fails:

```
fail :: String → Maybe a
fail _ = Nothing
```

For example, the clause for *eval* (*If x y z*) desugars into the following:

```
eval (If x y z) = eval x ≫= λv → case v of B b → if b then eval y else eval z
                                          _   → Nothing
```

With the revised *eval* function, *Add* (*Val* (*B True*)) (*Val* (*N* 9)) now evaluates to *Nothing*: The first argument of *Add* evaluates to *B True* and thus the pattern match with *N m* fails, resulting in the value *Nothing*, which in turn is propagated according to the definition of ≫=.

2.1 Compiler Correctness

Our goal is to calculate a function $compile :: Expr \to Code$ that translates expressions into a target language *Code*. But the compiler calculation process not only produces a compiler; it also produces the syntax and the semantics of the target language [Meijer 1992]. We only assume that the compiler targets a stack machine, whose semantics is given by a function $exec :: Code \to Stack \to Maybe\ Stack$, where $Stack = [Value]$. Like the source language semantics, the semantics of the target language also uses the *Maybe* monad to account for unsafe behaviour.

All missing components – *compile*, *exec*, and *Code* – will be derived by the calculation process. But before we formulate the correctness property of the compiler, we generalise the function *compile* so that it takes an additional code argument that is meant to be inserted after the compiled code. This will significantly simplify the calculation process [Bahr and Hutton 2015]. Using this idea, the specification for the generalised compiler $comp :: Expr \to Code \to Code$ is as follows:

$$\mathbf{do}\ v \leftarrow eval\ e;\ exec\ c\ (v : s)\ =\ exec\ (comp\ e\ c)\ s \tag{1}$$

That is, compiling an expression and then executing the resulting code together with some additional code c gives the same result as executing c with the value of the expression on top of the stack. Both sides of the equation are of type *Maybe Stack*, thus accounting for the fact that both the interpreter *eval* and the stack machine *exec* may have unsafe behaviour.

We could now use specification (1) to calculate the compiler using monadic reasoning in the style of Bahr and Hutton [2022]. However, this specification may be too strict. In general, we might want to allow the compiler to exploit unsafe behaviour to perform optimisations. Leroy [2009] expresses this intuition of the relation between a source program S and the corresponding compiled program C as the following *behaviour refinement*:

$$\text{If Safe}(S),\ \text{then } \forall B.\ C \Downarrow B \implies S \Downarrow B \tag{2}$$

That is, given a safe source program, any behaviour B exhibited by the compiled program needs to be exhibited by the source program. A program S is safe if it has no wrong behaviours, i.e. $S \Downarrow B$ implies $B \notin \text{Wrong}$, for some set Wrong of wrong behaviours. For example, for our expression language, $\text{Wrong} = \{Nothing\}$. Under the assumption that both the source language and the target language are deterministic, Leroy strengthens this specification into a form that is easier to prove:

$$\forall B \notin \text{Wrong}.\ S \Downarrow B \implies C \Downarrow B \tag{3}$$

Translating this to our setting, we obtain the following relaxation of our compiler specification (1):

$$\mathbf{do}\ v \leftarrow eval\ e;\ exec\ c\ (v : s) = Just\ w \quad\implies\quad exec\ (comp\ e\ c)\ s = Just\ w \tag{4}$$

To perform calculations, we rephrase specification (4) into an inequality so that we can use equational reasoning. We achieve this by defining a suitable relation $\mathrel{\perp\!\!=}$ on *Maybe* types:

$$\begin{aligned} Nothing &\mathrel{\perp\!\!=} p && \text{for any } p :: Maybe\ a && (\perp\!\!=\text{-}Nothing)\\ Just\ v &\mathrel{\perp\!\!=} Just\ v && \text{for any } v :: a. && (\perp\!\!=\text{-}Just) \end{aligned}$$

Using this relation on *Maybe*, the above specification (4) can now be formulated equivalently as

$$\mathbf{do}\ v \leftarrow eval\ e;\ exec\ c\ (v : s)\ \mathrel{\perp\!\!=}\ exec\ (comp\ e\ c)\ s \tag{5}$$

Since $\mathrel{\perp\!\!=}$ is a preorder, i.e. it is reflexive and transitive, and is a congruence for $\gg\!=$, we can use the above specification as the basis for the compiler calculation technique of Bahr and Hutton [2022]. The crucial property of $\mathrel{\perp\!\!=}$ that will allow us to essentially ignore unsafe behaviour during

the compiler calculation is the fact that *Nothing* is the least element w.r.t. $\mathrel{\bot}=$ according to the defining clause ($\mathrel{\bot}$=-*Nothing*). In addition, we will make extensive use of the monad laws:

$$\begin{aligned} \mathit{return}\ x \gg\!= f &= f\ x && \text{(left identity)} \\ mx \gg\!= \mathit{return} &= mx && \text{(right identity)} \\ (mx \gg\!= f) \gg\!= g &= mx \gg\!= (\lambda x\,.\,(f\ x \gg\!= g)) && \text{(associativity)} \end{aligned}$$

2.2 Compiler Calculation

We can now calculate the desired compiler for the source language. To this end, we prove specification (5) by induction on the structure of the expression e, and we do so before we even have definitions of *comp*, *exec*, and *Code*. As we shall see, the definitions of the missing components will naturally fall out of the calculation: For each case of e, we start on the left-hand side of the inequality and seek to transform it into the form $\mathit{exec}\ c'\ s$ for some code c', thus proving

$$\mathbf{do}\ v \leftarrow \mathit{eval}\ e\,;\ \mathit{exec}\ c\ (v : s) \ \mathrel{\bot}= \ \mathit{exec}\ c'\ s$$

We can then set $\mathit{comp}\ e\ c = c'$ as the definition of the compiler for e, which completes the proof for this case. Moreover, during the calculation, we will discover new clauses for the definition of the stack machine *exec* along with new constructors for *Code*. We start with the case for $e = \mathit{Val}\ v$:

$$\begin{aligned} & \mathbf{do}\ w \leftarrow \mathit{eval}\ (\mathit{Val}\ v)\,;\ \mathit{exec}\ c\ (w : s) \\ = \quad & \{\ \text{definition of } \mathit{eval}\ \} \\ & \mathbf{do}\ w \leftarrow \mathit{return}\ v\,;\ \mathit{exec}\ c\ (w : s) \\ = \quad & \{\ \text{left identity monad law}\ \} \\ & \mathit{exec}\ c\ (v : s) \end{aligned}$$

In these first two steps, we simply apply the definition of *eval* and simplify the term using the left identity law of the *Maybe* monad. But then we appear to be stuck as we cannot further simplify $\mathit{exec}\ c\ (v : s)$. Our final goal is a term of the form $\mathit{exec}\ c'\ s$ for some c'. That is, given the current term $\mathit{exec}\ c\ (v : s)$, we have to solve the inequality $\mathit{exec}\ c'\ s \mathrel{=\!\!\bot} \mathit{exec}\ c\ (v : s)$ for some c'. Here we use the notation $=\!\!\bot$ for the inverse relation of $\mathrel{\bot}=$. We can solve this inequality by strengthening it to an equality $\mathit{exec}\ c'\ s = \mathit{exec}\ c\ (v : s)$ and solving this equation by picking a suitable c' so that the equation holds *by definition*. We can view this equation as a definitional clause for *exec* if c' is a pattern and all variables on the right-hand side of the equation also occur on the left-hand side. That means, c' must be a pattern that includes v and c, which we can achieve by introducing a new constructor $\mathit{PUSH} :: \mathit{Value} \to \mathit{Code} \to \mathit{Code}$ and taking $c' = \mathit{PUSH}\ v\ c$. That is, we have discovered the first instruction of the stack machine and its semantics: $\mathit{exec}\ (\mathit{PUSH}\ v\ c)\ s = \mathit{exec}\ c\ (v : s)$. Equipped with this observation, we can resume our calculation and finish it:

$$\begin{aligned} & \mathit{exec}\ c\ (v : s) \\ = \quad & \{\ \text{define: } \mathit{exec}\ (\mathit{PUSH}\ v\ c)\ s = \mathit{exec}\ c\ (v : s)\ \} \\ & \mathit{exec}\ (\mathit{PUSH}\ v\ c)\ s \end{aligned}$$

Since we arrived at a term of the desired form $\mathit{exec}\ c'\ s$, we can read off the first clause for *comp*:

$$\mathit{comp}\ (\mathit{Val}\ v)\ c = \mathit{PUSH}\ v\ c$$

We proceed with the case for $e = \mathit{Add}\ x\ y$. Starting with the left-hand side term of specification (5), our goal is to manipulate the term into the form $\mathit{exec}\ c'\ s$. On the way to that goal, we aim to apply the induction hypotheses for x and y. For y, the induction hypothesis is as follows:

$$\mathbf{do}\ v \leftarrow \mathit{eval}\ y\,;\ \mathit{exec}\ c'\ (v : s') \ \mathrel{\bot}= \ \mathit{exec}\ (\mathit{comp}\ y\ c')\ s' \qquad \text{for all } c', s' \qquad \text{(IH-}y\text{)}$$

As in the previous case, we first apply the definition of *eval* followed by the monad laws:

$$
\begin{aligned}
&\mathbf{do}\ v \leftarrow \mathit{eval}\ (\mathit{Add}\ x\ y)\,;\ \mathit{exec}\ c\ (v : s)\\
=\quad &\{\,\text{definition of } \mathit{eval}\,\}\\
&\mathbf{do}\ v \leftarrow \mathbf{do}\ \{N\ m \leftarrow \mathit{eval}\ x\,;\ N\ n \leftarrow \mathit{eval}\ y\,;\ \mathit{return}\ (N\ (m + n))\}\,;\ \mathit{exec}\ c\ (v : s)\\
=\quad &\{\,\text{left identity \& associativity monad laws}\,\}\\
&\mathbf{do}\ N\ m \leftarrow \mathit{eval}\ x\,;\ N\ n \leftarrow \mathit{eval}\ y\,;\ \mathit{exec}\ c\ (N\ (m + n) : s)
\end{aligned}
$$

We appear to be stuck again. Our goal is to apply the induction hypothesis for y, i.e. we seek a term matching the left-hand side of (IH-y). This means, we need to solve the following inequality:

$$\mathbf{do}\ v \leftarrow \mathit{eval}\ y\,;\ \mathit{exec}\ c'\ (v : s') =_\bot \mathbf{do}\ N\ n \leftarrow \mathit{eval}\ y\,;\ \mathit{exec}\ c\ (N\ (m + n) : s) \qquad \text{for some } c', s'$$

We won't be able to solve this inequality in one go as we did in the *Val* case, but we can solve a simpler inequality that gets us closer. We solve the inequality for the case where $v = N\ n$ so that both sides coincide on the initial part $\mathbf{do}\ N\ n \leftarrow \mathit{eval}\ y$. Thus, it remains to solve the following:

$$\mathit{exec}\ c'\ (N\ n : s') =_\bot \mathit{exec}\ c\ (N\ (m + n) : s) \qquad \text{for some } c', s'$$

As in the *Val* case, we solve this inequality by first strengthening it to an equality (i.e. changing $=_\bot$ to $=$) and then finding a suitable instantiation of its existentially quantified variables s' and c' so that the equation can be interpreted as a definition for *exec*. In particular, we need to ensure that each universally quantified variable has exactly one occurrence on the left-hand side. At the moment this is only the case for n, and thus we must instantiate s' and c' so that they include c, m, and s exactly once. Since m and s comprise data, they naturally fit into the stack s', whereas the code c naturally fits into the code c'. We achieve this by instantiating $s' = N\ m : s$ and by introducing a constructor $\mathit{ADD} :: \mathit{Code} \to \mathit{Code}$ so that we can instantiate $c' = \mathit{ADD}\ c$:

$$
\begin{aligned}
&\mathbf{do}\ N\ m \leftarrow \mathit{eval}\ x\,;\ N\ n \leftarrow \mathit{eval}\ y\,;\ \mathit{exec}\ c\ (N\ (m + n) : s)\\
=\quad &\{\,\text{define: } \mathit{exec}\ (\mathit{ADD}\ c)\ (N\ n : N\ m : s) = \mathit{exec}\ c\ (N\ (m + n) : s)\,\}\\
&\mathbf{do}\ N\ m \leftarrow \mathit{eval}\ x\,;\ N\ n \leftarrow \mathit{eval}\ y\,;\ \mathit{exec}\ (\mathit{ADD}\ c)\ (N\ n : N\ m : s)
\end{aligned}
$$

As anticipated, the relevant subterm $\mathbf{do}\ N\ n \leftarrow \mathit{eval}\ y\,;\ \mathit{exec}\ (\mathit{ADD}\ c)\ (N\ n : N\ m : s)$ still does not quite match the left-hand side of the induction hypothesis for y. To see why, let's unfold the syntactic sugar for the non-exhaustive pattern matching against $N\ n$:

$$
\begin{aligned}
&\mathbf{do}\ v \leftarrow \mathit{eval}\ y\,;\ \mathbf{case}\ v\ \mathbf{of}\ N\ n \to \mathit{exec}\ (\mathit{ADD}\ c)\ (N\ n : N\ m : s)\\
&\qquad\qquad\qquad\qquad\qquad\quad \_ \quad \to \mathit{Nothing}
\end{aligned}
$$

For this term to match the left-hand side of the induction hypothesis, we have to transform it so that both cases of the pattern matching expression produce the same term. That is readily achieved by appealing to the $_\bot{=}$-*Nothing* law to conclude that $\mathit{Nothing}\ {}_\bot{=}\ \mathit{exec}\ (\mathit{ADD}\ c)\ (v : N\ m : s)$:

$$
\begin{aligned}
&\mathbf{do}\ v \leftarrow \mathit{eval}\ y\,;\ \mathbf{case}\ v\ \mathbf{of}\ N\ n \to \mathit{exec}\ (\mathit{ADD}\ c)\ (N\ n : N\ m : s)\\
&\qquad\qquad\qquad\qquad\qquad\quad \_ \quad \to \mathit{Nothing}\\
{}_\bot{=}\quad &\{\ {}_\bot{=}\text{-}\mathit{Nothing}\ \text{law}\,\}\\
&\mathbf{do}\ v \leftarrow \mathit{eval}\ y\,;\ \mathbf{case}\ v\ \mathbf{of}\ N\ n \to \mathit{exec}\ (\mathit{ADD}\ c)\ (N\ n : N\ m : s)\\
&\qquad\qquad\qquad\qquad\qquad\quad \_ \quad \to \mathit{exec}\ (\mathit{ADD}\ c)\ (v : N\ m : s)\\
=\quad &\{\,\text{case analysis on } v\,\}\\
&\mathbf{do}\ v \leftarrow \mathit{eval}\ y\,;\ \mathit{exec}\ (\mathit{ADD}\ c)\ (v : N\ m : s)
\end{aligned}
$$

Using this idea, we can resume our calculation for *Add*:

$$
\begin{aligned}
&\mathbf{do}\ N\ m \leftarrow \mathit{eval}\ x\,;\ N\ n \leftarrow \mathit{eval}\ y\,;\ \mathit{exec}\ (\mathit{ADD}\ c)\ (N\ n : N\ m : s)\\
{}_\bot{=}\quad &\{\ {}_\bot{=}\text{-}\mathit{Nothing}\ \text{law}\,\}\\
&\mathbf{do}\ N\ m \leftarrow \mathit{eval}\ x\,;\ v \leftarrow \mathit{eval}\ y\,;\ \mathit{exec}\ (\mathit{ADD}\ c)\ (v : N\ m : s)
\end{aligned}
$$

$$
\begin{aligned}
&\ {}_{\perp}= \quad \{\ \text{induction hypothesis for } y\ \}\\
&\quad \mathbf{do}\ N\ m \leftarrow \mathit{eval}\ x\,;\ \mathit{exec}\ (\mathit{comp}\ y\ (\mathit{ADD}\ c))\ (N\ m : s)\\
&\ {}_{\perp}= \quad \{\ {}_{\perp}\text{=-}\mathit{Nothing}\ \text{law}\ \}\\
&\quad \mathbf{do}\ v \leftarrow \mathit{eval}\ x\,;\ \mathit{exec}\ (\mathit{comp}\ y\ (\mathit{ADD}\ c))\ (v : s)\\
&\ {}_{\perp}= \quad \{\ \text{induction hypothesis for } x\ \}\\
&\quad \mathit{exec}\ (\mathit{comp}\ x\ (\mathit{comp}\ y\ (\mathit{ADD}\ c)))\ s
\end{aligned}
$$

After having applied the induction hypothesis for y, we again appeal to the ${}_{\perp}$=-*Nothing* law to transform the term so that we can apply the induction hypothesis for x. The resulting term is of the desired form $\mathit{exec}\ c'\ s$, and we can therefore read off the next clause of the compiler:

$$\mathit{comp}\ (\mathit{Add}\ x\ y)\ c = \mathit{comp}\ x\ (\mathit{comp}\ y\ (\mathit{ADD}\ c))$$

In the final case, namely for $e = \mathit{If}\ x\ y\ z$, we can apply the lessons we learned in the first two cases to complete the calculation:

$$
\begin{aligned}
&\quad \mathbf{do}\ v \leftarrow \mathit{eval}\ (\mathit{If}\ x\ y\ z)\,;\ \mathit{exec}\ c\ (v : s)\\
&= \quad \{\ \text{definition of } \mathit{eval}\ \}\\
&\quad \mathbf{do}\ v \leftarrow \mathbf{do}\ \{B\ b \leftarrow \mathit{eval}\ x\,;\ \mathbf{if}\ b\ \mathbf{then}\ \mathit{eval}\ y\ \mathbf{else}\ \mathit{eval}\ z\}\,;\ \mathit{exec}\ c\ (v : s)\\
&= \quad \{\ \text{associativity monad law}\ \}\\
&\quad \mathbf{do}\ B\ b \leftarrow \mathit{eval}\ x\,;\ \mathbf{if}\ b\ \mathbf{then}\ (\mathbf{do}\ v \leftarrow \mathit{eval}\ y\,;\ \mathit{exec}\ c\ (v : s))\\
&\qquad\qquad\qquad\qquad\ \mathbf{else}\ \ (\mathbf{do}\ v \leftarrow \mathit{eval}\ z\,;\ \mathit{exec}\ c\ (v : s))\\
&\ {}_{\perp}= \quad \{\ \text{induction hypothesis for } y \text{ and } z\ \}\\
&\quad \mathbf{do}\ B\ b \leftarrow \mathit{eval}\ x\,;\ \mathbf{if}\ b\ \mathbf{then}\ \mathit{exec}\ (\mathit{comp}\ y\ c)\ s\\
&\qquad\qquad\qquad\qquad\ \mathbf{else}\ \ \mathit{exec}\ (\mathit{comp}\ z\ c)\ s\\
&= \quad \{\ \text{define: } \mathit{exec}\ (\mathit{JPC}\ c'\ c)\ (B\ b : s) = \mathbf{if}\ b\ \mathbf{then}\ \mathit{exec}\ c'\ s\ \mathbf{else}\ \mathit{exec}\ c\ s\ \}\\
&\quad \mathbf{do}\ B\ b \leftarrow \mathit{eval}\ x\,;\ \mathit{exec}\ (\mathit{JPC}\ (\mathit{comp}\ y\ c)\ (\mathit{comp}\ z\ c))\ (B\ b : s)\\
&\ {}_{\perp}= \quad \{\ {}_{\perp}\text{=-}\mathit{Nothing}\ \text{law}\ \}\\
&\quad \mathbf{do}\ v \leftarrow \mathit{eval}\ x\,;\ \mathit{exec}\ (\mathit{JPC}\ (\mathit{comp}\ y\ c)\ (\mathit{comp}\ z\ c))\ (v : s)\\
&\ {}_{\perp}= \quad \{\ \text{induction hypothesis for } x\ \}\\
&\quad \mathit{exec}\ (\mathit{comp}\ x\ (\mathit{JPC}\ (\mathit{comp}\ y\ c)\ (\mathit{comp}\ z\ c)))\ s
\end{aligned}
$$

We have thus found the final clause of the compiler definition:

$$\mathit{comp}\ (\mathit{If}\ x\ y\ z)\ c = \mathit{comp}\ x\ (\mathit{JPC}\ (\mathit{comp}\ y\ c)\ (\mathit{comp}\ z\ c))$$

To complete the compiler calculation, we consider the top-level compiler $\mathit{compile} :: \mathit{Expr} \to \mathit{Code}$, whose correctness is captured by the following property:

$$\mathbf{do}\ v \leftarrow \mathit{eval}\ e\,;\ \mathit{return}\ (v : s) \quad {}_{\perp}= \quad \mathit{exec}\ (\mathit{compile}\ e)\ s$$

To calculate the definition of *compile*, we again start on the left-hand side and seek to manipulate the term into the form $\mathit{exec}\ c'\ s$ so that we can define $\mathit{compile}\ e = c'$. However, this time no induction is necessary:

$$
\begin{aligned}
&\quad \mathbf{do}\ v \leftarrow \mathit{eval}\ e\,;\ \mathit{return}\ (v : s)\\
&= \quad \{\ \text{define: } \mathit{exec}\ \mathit{HALT}\ s = \mathit{return}\ s\ \}\\
&\quad \mathbf{do}\ v \leftarrow \mathit{eval}\ e\,;\ \mathit{exec}\ \mathit{HALT}\ (v : s)\\
&\ {}_{\perp}= \quad \{\ \text{correctness property (5) of } \mathit{comp}\ \}\\
&\quad \mathit{exec}\ (\mathit{comp}\ e\ \mathit{HALT})\ s
\end{aligned}
$$

This completes the calculation. In summary, we have calculated the definitions in Figure 1. Note that the equations we derived for *exec* are not exhaustive. In order to make *exec* total, we simply

Target language

```
data Code = PUSH Value Code | ADD Code
          | JPC Code Code | HALT
```

Compiler

```
compile :: Expr → Code
compile e = comp e HALT

comp :: Expr → Code → Code
comp (Val v)     c = PUSH v c
comp (Add x y)   c = comp x (comp y (ADD c))
comp (If x y z)  c = comp x (JPC (comp y c) (comp z c))
```

Stack machine

```
type Stack = [Value]

exec :: Code → Stack → Maybe Stack
exec (PUSH v c) s = exec c (v : s)
exec (ADD c) (N n : N m : s)
  = exec c (N (m + n) : s)
exec (JPC c' c) (B b : s)
  = if b then exec c' s else exec c s
exec HALT s       = return s
exec _ _          = Nothing
```

Fig. 1. Compiler and stack machine for the simple expression language with conditionals.

add a catch-all clause that returns *Nothing*, which we are allowed to freely choose as the calculation does not depend on it.

By using a partial compiler specification that uses the relation $\mathrel{\bot}=$ instead of strict equality, we were able to focus the calculation on the safe behaviour of the source language. The use of the $\mathrel{\bot}$*=-Nothing* law allowed us to ignore unsafe behaviour.

For this rather simple language, the use of the weaker specification was not strictly necessary. We would also be able to calculate the same compiler and stack machine using the stronger specification, albeit at the cost of a more complicated calculation. However, this only works since the language has no side effects. In the next section, we will see that as soon as the source language has side effects, e.g. non-termination, the transition to a partial compiler specification is essential.

3 Choice Trees

In section 2, we used the *Maybe* monad to express unsafe behaviour of our source and target language. This simple approach is only suitable for languages without observable effects, i.e. if the only observable behaviour is the final result of evaluation. To generalise this calculation approach to account for effects such as non-termination, non-determinism, and algebraic effects, we replace the *Maybe* monad with *choice trees* [Chappe et al. 2025]. As with *Maybe*, choice trees serve as the common semantic framework to describe both the source and the target language semantics.

In this section, we recall the basic definitions of choice trees and demonstrate the shortcomings of using standard bisimilarity for equational reasoning in the presence of unsafe behaviour. We then introduce our solution to this issue – namely skew bisimilarity – in section 4.

3.1 Syntax

The type of choice trees *CTree e a* classifies non-deterministic computations that return values of type *a* and that may use algebraic effects described by the type function $e :: \mathit{Type} \to \mathit{Type}$:

```
data CTree e a where
  Now  :: a → CTree e a
  (⊕)  :: CTree e a → CTree e a → CTree e a
  Zero :: CTree e a
  Eff  :: e b → (b → CTree e a) → CTree e a
  Later :: ∞ (CTree e a) → CTree e a
```

Now v is a computation that returns the value v without performing any effects, $p \oplus q$ is a computation that makes a non-deterministic choice between two computations p and q, *Zero*

is a computation that has terminated, *Eff o c* is a computation that first performs an effectful computation *o* and then passes the result of that into a continuation *c*, and *Later p* behaves like *p* but allows us to express non-terminating computations as we will explain further below.

As an example, we construct an effectful operation that prints an integer. This can be achieved by first defining a type function *PrintEff* that provides a single constructor called *PrintInt*, which is then used to define a *print* function:

```
data PrintEff a where
  PrintInt :: Int → PrintEff ()

print :: Int → CTree PrintEff ()
print n = Eff (PrintInt n) Now
```

CTree is an inductive type, defined mutually recursively with a coinductive type denoted by ∞ (*CTree e a*). Here we adopt the ∞ notation for mixed inductive-coinductive types in Agda [Danielsson and Altenkirch 2010], but to reduce clutter we assume that conversions *Delay* :: *CTree e a* → ∞ (*CTree e a*) and *Force* :: ∞ (*CTree e a*) → *CTree e a* are inserted implicitly in the appropriate places as in Idris [Brady 2017]. In short, this means that *Later* is a coinductive constructor whereas all other constructors of *CTree* are inductive. That is, a value of type *CTree e a* is a potentially infinite tree with nodes labelled by the constructors *Now*, $\oplus$, *Zero*, *Eff*, and *Later*, such that every infinite path from the root must contain infinitely many nodes labelled *Later*. For example, a computation that never terminates can be defined as follows:

```
never :: CTree e a
never = Later never
```

Despite being non-terminating, this definition is total because the recursive call is guarded by *Later*, and systems such as Agda will accept it.

Choice trees form a monad, with *return* and $\gg=$ defined as follows:

```
return :: a → CTree e a
return = Now

(>>=) :: CTree e a → (a → CTree e b) → CTree e b
Now v   >>= f = f v
(p ⊕ q) >>= f = (p >>= f) ⊕ (q >>= f)
Zero    >>= f = Zero
Eff o c >>= f = Eff o (λi → c i >>= f)
Later p >>= f = Later (p >>= f)
```

Similarly to the *Maybe* monad, we will use *do notation* in place of explicit applications of $\gg=$.

We can use an algebraic effect type *Stuck* to represent unsafe behaviour:

```
data Stuck a where
  Stuck :: Stuck Void

stuck :: CTree Stuck a
stuck = Eff Stuck elimVoid
```

where *Void* is the empty data type and *elimVoid* :: *Void* → *a* is its eliminator. We use the name *Stuck*, as we can think of this effect as representing a computation that is stuck. Like the *Maybe* monad, we can make *CTree Stuck* an instance of the *MonadFail* type class by using *stuck* to signal failure:

```
fail :: String → CTree Stuck a
fail _ = stuck
```

We could now revise the evaluator from section 2 to use *CTree Stuck* instead of *Maybe*. In fact, this new definition of *eval* using *CTree Stuck* would remain exactly the same and only its type signature would change. But *CTree Stuck* provides a much richer semantic framework that allows us to express effectful computations. To demonstrate this, we extend the expression language with a simple algebraic effect and a simple non-deterministic operation:

$$\frac{}{\mathit{Now}\ v \overset{\langle v \rangle}{\Longrightarrow} \mathit{Zero}} \qquad \frac{p \overset{l}{\Longrightarrow} p'}{p \oplus q \overset{l}{\Longrightarrow} p'} \qquad \frac{q \overset{l}{\Longrightarrow} q'}{p \oplus q \overset{l}{\Longrightarrow} q'} \qquad \frac{}{\mathit{Eff}\ o\ c \overset{\uparrow o}{\Longrightarrow} c} \qquad \frac{}{c \overset{\downarrow i}{\Longrightarrow} c\ i} \qquad \frac{}{\mathit{Later}\ p \overset{\tau}{\Longrightarrow} p}$$

Fig. 2. Transition semantics for choice trees.

$$\mathbf{data}\ \mathit{Value} = \mathit{B}\ \mathit{Bool} \mid \mathit{N}\ \mathit{Int}$$
$$\mathbf{data}\ \mathit{Expr} = \mathit{Val}\ \mathit{Value} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr} \mid \mathit{If}\ \mathit{Expr}\ \mathit{Expr}\ \mathit{Expr} \mid \mathit{Print}\ \mathit{Expr} \mid \mathit{Flip}$$

Informally, *Print* prints its argument, and *Flip* non-deterministically evaluates to true or false.

The evaluator for this language needs access to both the *Stuck* and the *PrintEff* effect. To this end, we define the coproduct of two effect types as follows:

$$\mathbf{data}\ (e \uplus f)\ a = \mathit{Inl}\ (e\ a) \mid \mathit{Inr}\ (f\ a)$$

Using *data types à la carte* [Swierstra 2008], we can define a type class $\prec$ that captures the fact that an effect type is contained in another effect type, e.g. $\mathit{Stuck} \prec \mathit{Stuck} \uplus \mathit{PrintEff}$. In particular, $\prec$ provides an injection function $\mathit{inj} :: e \prec f \Rightarrow e\ a \to f\ a$, which allows us to generalise the *print* and *stuck* effects, as well as the *fail* method of the *MonadFail* instance:

$$\begin{aligned}
&\mathit{stuck} :: \mathit{Stuck} \prec e \Rightarrow \mathit{CTree}\ e\ a && \mathit{print} :: \mathit{PrintEff} \prec e \Rightarrow \mathit{Int} \to \mathit{CTree}\ e\ () \\
&\mathit{stuck} = \mathit{Eff}\ (\mathit{inj}\ \mathit{Stuck})\ \mathit{elimVoid} && \mathit{print}\ n = \mathit{Eff}\ (\mathit{inj}\ (\mathit{PrintInt}\ n))\ \mathit{Now} \\
&\mathit{fail} :: \mathit{Stuck} \prec e \Rightarrow \mathit{String} \to \mathit{CTree}\ e\ a \\
&\mathit{fail}\ _ = \mathit{stuck}
\end{aligned}$$

Finally, we can define the semantics of the expression language extended with *Print* and *Flip*:

$$\begin{aligned}
&\mathit{eval} :: \mathit{Expr} \to \mathit{CTree}\ (\mathit{Stuck} \uplus \mathit{PrintEff})\ \mathit{Value} \\
&\mathit{eval}\ (\mathit{Val}\ v) && = \mathit{return}\ v \\
&\mathit{eval}\ (\mathit{Add}\ x\ y) && = \mathbf{do}\ N\ m \leftarrow \mathit{eval}\ x;\ N\ n \leftarrow \mathit{eval}\ y;\ \mathit{return}\ (N\ (m+n)) \\
&\mathit{eval}\ (\mathit{If}\ x\ y\ z) && = \mathbf{do}\ B\ b \leftarrow \mathit{eval}\ x;\ \mathbf{if}\ b\ \mathbf{then}\ \mathit{eval}\ y\ \mathbf{else}\ \mathit{eval}\ z \\
&\mathit{eval}\ (\mathit{Print}\ x) && = \mathbf{do}\ N\ n \leftarrow \mathit{eval}\ x;\ \mathit{print}\ n;\ \mathit{return}\ (N\ n) \\
&\mathit{eval}\ \mathit{Flip} && = \mathit{return}\ (B\ \mathit{True}) \oplus \mathit{return}\ (B\ \mathit{False})
\end{aligned}$$

To calculate a compiler for this extended expression language, we generalise the relation $\perp\!=$ from *Maybe* to choice trees. To this end, we first present the semantics of choice trees and the resulting notion of bisimilarity.

3.2 Semantics

Following Bahr and Hutton [2023], the semantics of choice trees is given by the labelled transition system defined in Figure 2. A state in this labelled transition system is either a choice tree p::$\mathit{CTree}\ e\ a$ or a continuation $c :: b \to \mathit{CTree}\ e\ a$ that is waiting for some external input of type b in response to an immediately preceding effect of type $e\ b$. The labels of the labelled transition system take on one of four forms: a silent transition τ, a return value $\langle v \rangle$ with $v :: a$, an effect $\uparrow o$ with $o :: e\ b$ for some type b, or an input $\downarrow i$ with $i :: b$ for some type b.

As an example, we take a closer look at the labelled transition system of the choice tree

$$\begin{aligned}
&p :: \mathit{CTree}\ (\mathit{Stuck} \uplus \mathit{PrintEff})\ \mathit{Int} \\
&p = \mathit{stuck} \oplus (\mathit{print}\ 1 \gg\!= \lambda() \to \mathit{return}\ 2)
\end{aligned}$$

which after unfolding the definitions of *stuck*, *print*, *return*, and $\gg\!=$ expands to the following:

$$p = \mathit{Eff}\ (\mathit{Inl}\ \mathit{Stuck})\ \mathit{elimVoid} \oplus \mathit{Eff}\ (\mathit{Inr}\ (\mathit{PrintInt}\ 1))\ (\lambda() \to \mathit{Now}\ 2)$$

Because of the choice operator $\oplus$, p has two possible transition sequences: It can either get stuck,

$$p \xrightarrow{\uparrow \mathit{Inl}\ \mathit{Stuck}} \mathit{elimVoid}$$

or it can print 1, consume the resulting unit value (), and finally return 2:

$$p \xrightarrow{\uparrow \mathit{Inr}\ (\mathit{PrintInt}\ 1)} \lambda() \to \mathit{Now}\ 2 \xrightarrow{\downarrow()} \mathit{Now}\ 2 \xrightarrow{\langle 2 \rangle} \mathit{Zero}$$

Both $\mathit{elimVoid} :: \mathit{Void} \to \mathit{CTree}\ (\mathit{Stuck} \uplus \mathit{PrintEff})\ \mathit{Int}$ and $\mathit{Zero} :: \mathit{CTree}\ (\mathit{Stuck} \uplus \mathit{PrintEff})\ \mathit{Int}$ are terminal, i.e. there are no transitions of the form $\mathit{elimVoid} \xrightarrow{l} q$ or $\mathit{Zero} \xrightarrow{l} q$.

3.3 Bisimilarity

Because the semantics of choice trees is defined using a labelled transition system, the notion of bisimilarity $\cong$ can be defined in the standard way. Concretely, $\cong$ is the largest relation on choice trees (and continuations) that satisfies the following two properties:

(1) If $p \cong q$ and $p \xrightarrow{l} p'$, then there is some $q \xrightarrow{l} q'$ with $p' \cong q'$.
(2) If $p \cong q$ and $q \xrightarrow{l} q'$, then there is some $p \xrightarrow{l} p'$ with $p' \cong q'$.

The type of choice trees forms a monad modulo bisimilarity, i.e. it satisfies the following laws:

$$\mathit{return}\ x \gg\!= f \ \cong\ f\ x \qquad \text{(left identity)}$$
$$p \gg\!= \mathit{return} \ \cong\ p \qquad \text{(right identity)}$$
$$(p \gg\!= f) \gg\!= g \ \cong\ p \gg\!= \lambda x \to (f\ x \gg\!= g) \qquad \text{(associativity)}$$

Moreover, all operations on choice trees we consider in this article satisfy congruence laws with respect to bisimilarity. For the monadic bind operator, the congruence law is stated as follows:

$$p \gg\!= f \ \cong\ q \gg\!= g \qquad \text{if } p \cong q \text{ and } f\ x \cong g\ x \text{ for all } x \qquad (\gg\!=\text{-congruence})$$

These laws will suffice for the simple language we are considering. For a more comprehensive overview of the equational theory of choice trees, we refer the interested reader to Bahr and Hutton [2023]. But as an illustration, we take a closer look at $\mathit{Zero} \oplus p \cong p$, the left unit law for $\oplus$. When using a small-step operational semantics it is customary to consider stuck terms – i.e. terms without outgoing transitions – as unsafe. To prove that a language is safe, one then usually proves progress and preservation properties, which imply that no stuck term is reachable. Even though *Zero* is a 'stuck' choice tree in the sense that it has no outgoing transitions, the left unit law for $\oplus$ shows that *Zero* cannot be used to indicate unsafe behaviour in the context of non-determinism. By contrast, we don't have that $\mathit{stuck} \oplus p \cong p$ holds for all p.

3.4 Compiler Calculation

We now aim to calculate a compiler $\mathit{comp} :: \mathit{Expr} \to \mathit{Code} \to \mathit{Code}$ that targets a stack machine

$\mathit{exec} :: \mathit{Code} \to \mathit{Stack} \to \mathit{CTree}\ (\mathit{Stuck} \uplus \mathit{PrintEff})\ \mathit{Stack}$

With the help of bisimilarity, we can state the compiler specification as follows:

$$\mathbf{do}\ v \leftarrow \mathit{eval}\ e;\ \mathit{exec}\ c\ (v : s) \ \cong\ \mathit{exec}\ (\mathit{comp}\ e\ c)\ s \qquad (6)$$

Similarly to our first specification (1) in section 2.1, this property is too strong as it requires the compiler to produce code with the same behaviour as the source expression – even if the expression is ill-typed. We will therefore revise this specification in section 4. But before we do so, we calculate a compiler with this overly strong specification to observe the outcome it produces.

We proceed by induction on e and try to manipulate the left-hand side of the specification (6) into the form $exec\ c'\ s$ so that we can conclude $comp\ e\ c = c'$. The case for $e = Val\ v$ is straightforward and is essentially the same as the calculation in section 2. The case for $e = Add\ x\ y$ starts similarly to the calculation in section 2 as well:

$$
\begin{aligned}
&\mathbf{do}\ v \leftarrow eval\ (Add\ x\ y)\,;\ exec\ c\ (v : s)\\
=\quad &\{\,\text{definition of } eval\,\}\\
&\mathbf{do}\ v \leftarrow \mathbf{do}\ \{N\ m \leftarrow eval\ x\,;\ N\ n \leftarrow eval\ y\,;\ return\ (N\ (m + n))\}\,;\ exec\ c\ (v : s)\\
\cong\quad &\{\,\text{left identity \& associativity monad laws}\,\}\\
&\mathbf{do}\ N\ m \leftarrow eval\ x\,;\ N\ n \leftarrow eval\ y\,;\ exec\ c\ (N\ (m + n) : s)\\
=\quad &\{\,\text{define: } exec\ (ADD\ c)\ (N\ n : N\ m : s) = exec\ c\ (N\ (m + n) : s)\,\}\\
&\mathbf{do}\ N\ m \leftarrow eval\ x\,;\ N\ n \leftarrow eval\ y\,;\ exec\ (ADD\ c)\ (N\ n : N\ m : s)
\end{aligned}
$$

Next we aim to manipulate the term so that we can apply the induction hypothesis for y. In the calculation in section 2, we used the $\bot$=-*Nothing* law for that purpose. Since we don't have a law of that form at our disposal here, we have to introduce a new equation for the stack machine instead:

$$
\begin{aligned}
&\mathbf{do}\ N\ m \leftarrow eval\ x\,;\ N\ n \leftarrow eval\ y\,;\ exec\ (ADD\ c)\ (N\ n : N\ m : s)\\
=\quad &\{\,\text{define: } exec\ (ADD\ c)\ (B\ b : s) = stuck\,\}\\
&\mathbf{do}\ N\ m \leftarrow eval\ x\,;\ v \leftarrow eval\ y\,;\ exec\ (ADD\ c)\ (v : N\ m : s)\\
\cong\quad &\{\,\text{induction hypothesis for } y\,\}\\
&\mathbf{do}\ N\ m \leftarrow eval\ x\,;\ exec\ (comp\ y\ (ADD\ c))\ (N\ m : s)
\end{aligned}
$$

That is, instead of relying on a law like $\bot$=-*Nothing* to rewrite *stuck* to $exec\ (ADD\ c)\ (B\ b : s)$, we have to make *stuck* equal to $exec\ (ADD\ c)\ (B\ b : s)$ by defining the semantics of *ADD* accordingly.

We are now in the same situation as above: We seek to manipulate the term so that we can apply the induction hypothesis for x. Again, we can achieve this by introducing a new equation for *exec*. But now this requires the insertion of a new instruction *ISN*, which checks whether the top of the stack is a number and gets stuck if this check fails:

$$
\begin{aligned}
&\mathbf{do}\ N\ m \leftarrow eval\ x\,;\ exec\ (comp\ y\ (ADD\ c))\ (N\ m : s)\\
=\quad &\{\,\text{define: } exec\ (ISN\ c)\ (B\ \_ : s) = stuck;\ exec\ (ISN\ c)\ (N\ n : s) = exec\ c\ (N\ n : s)\,\}\\
&\mathbf{do}\ v \leftarrow eval\ x\,;\ exec\ (ISN\ (comp\ y\ (ADD\ c)))\ (v : s)\\
\cong\quad &\{\,\text{induction hypothesis for } x\,\}\\
&exec\ (comp\ x\ (ISN\ (comp\ y\ (ADD\ c))))\ s
\end{aligned}
$$

This completes the calculation for the *Add* case. But the compiler now produces code with an additional *ISN* instruction for each *ADD* instruction:

$$comp\ (Add\ x\ y)\ c = comp\ x\ (ISN\ (comp\ y\ (ADD\ c)))$$

Intuitively speaking, the *ISN* instruction ensures that execution gets stuck as soon as the code for x produces a value of the wrong type, which matches the behaviour of $eval\ (Add\ x\ y)$ exactly. Without *ISN*, execution would only get stuck when trying to perform the *ADD* instruction *after* the code for y has been executed, which is observably different from $eval\ (Add\ x\ y)$ if y has side effects. However, if we are only interested in *safe* behaviours, we should be able to safely ignore such differences. We will explore this observation further with the help of an example in section 4.2.

We abort the calculation here and instead seek to revise the specification with the aim of avoiding instructions like *ISN*. To this end, we introduce a weaker bisimilarity relation $\bot\cong$ that corresponds to the $\bot=$ relation on *Maybe* and which satisfies a law corresponding to the $\bot$=-*Nothing* law that we were sorely missing in the above calculation. While this weakens the compiler correctness property, we will still obtain the strong correctness property (6) for well-typed expressions e.

4 Skew Bisimilarity

Our goal is to weaken the bisimilarity relation $\cong$ to account for unsafe behaviour in choice trees with an effect signature e containing *Stuck*. For notational convenience, we use the type shorthand $CTree_\bot$ defined below, and we refer to its elements as *partial choice trees*.

type $CTree_\bot\ e\ a = CTree\ (Stuck \uplus e)\ a$

We say that a partial choice tree p is *locally safe* if there is no transition $p \xrightarrow{\uparrow Inl\ Stuck} p'$. We then define *skew bisimilarity*, denoted $\mathrel{_\bot\cong}$, as the largest relation that satisfies the following:

(1) If $p \mathrel{_\bot\cong} q$, p is locally safe, and $p \xRightarrow{l} p'$, then there is some $q \xRightarrow{l} q'$ with $p' \mathrel{_\bot\cong} q'$.

(2) If $p \mathrel{_\bot\cong} q$, p is locally safe, and $q \xRightarrow{l} q'$, then there is some $p \xRightarrow{l} p'$ with $p' \mathrel{_\bot\cong} q'$.

That is, $p \mathrel{_\bot\cong} q$ vacuously holds if p is not locally safe. In particular, this means that skew bisimilarity satisfies the following additional law, which corresponds to the $_\bot$=-*Nothing* law for the relation $_\bot=$ on the *Maybe* monad and allows us to discard unsafe behaviour in compiler calculations:

$$stuck \mathrel{_\bot\cong} p \quad \text{for all } p \qquad (_\bot\cong\text{-}stuck)$$

Additionally, to help us relate bisimilarity and skew bisimilarity, we define the notion of *safe choice trees*: A partial choice tree p is *safe* if no transition sequence starting from p contains a transition labelled $\uparrow$ *Inl Stuck*. More formally, we can define safety as the largest predicate on partial choice trees (and continuations) with the following property:

$$\text{If } p \text{ is safe, then } p \text{ is locally safe and any } q \text{ with } p \xRightarrow{l} q \text{ is safe.}$$

Bisimilarity $\cong$ and skew bisimilarity $\mathrel{_\bot\cong}$ coincide if the left-hand side is safe:

Proposition 1. *Let $p, q :: CTree_\bot\ e\ a$ be two partial choice trees.*

(i) If $p \cong q$, then $p \mathrel{_\bot\cong} q$. *(ii) If p is safe and $p \mathrel{_\bot\cong} q$, then $p \cong q$.*

As a consequence, the equational laws of choice trees (see e.g. Bahr and Hutton [2023]) also hold for skew bisimilarity. In addition, all congruence laws hold for skew bisimilarity as well. Taken together, this makes skew bisimilarity a suitable choice for calculating compilers.

4.1 Compiler Calculation

In this section, we use skew bisimilarity to calculate a compiler for the arithmetic language extended with printing and non-determinism from section 3. Similarly to the aborted calculation in section 3.4, we aim to calculate a compiler $comp :: Expr \to Code \to Code$ for a stack machine $exec :: Code \to Stack \to CTree_\bot\ PrintEff\ Stack$. However, this time we weaken the compiler correctness specification by replacing bisimilarity with skew bisimilarity:

$$\mathbf{do}\ v \leftarrow eval\ e;\ exec\ c\ (v : s) \mathrel{_\bot\cong} exec\ (comp\ e\ c)\ s \tag{7}$$

We can now proceed with the calculation of the compiler by proving the above inequality by induction on the structure of e. In fact, the cases for $e = Val\ v$, $e = Add\ x\ y$, and $e = If\ x\ y\ z$ proceed in the same fashion as in section 2. Instead of the monad laws and the $_\bot$=-*Nothing* law for the *Maybe* monad, the calculation uses the monad laws and the $_\bot\cong$-*stuck* law of choice trees, respectively. The calculations for the remaining two cases are shown in Figure 3. The calculation for the case $e = Print\ x$ follows the same pattern as the *Add* case. For the case $e = Flip$, we use the fact that, by definition, $\gg$= distributes over $\oplus$ and then apply the left identity monad law. Apart from that, the calculation proceeds similarly to the *Val* case. From the calculation of these two cases, we can read off the definitions $comp\ (Print\ x)\ c = comp\ x\ (PRINT\ c)$ and $comp\ Flip\ c = FLIP\ c$.

$$
\begin{array}{ll}
& \mathbf{do}\ v \leftarrow \mathit{eval}\ (\mathit{Print}\ x)\,;\ \mathit{exec}\ c\ (v : s) \\
= & \quad \{\ \text{definition of } \mathit{eval}\ \} \\
& \mathbf{do}\ v \leftarrow \mathbf{do}\ \{N\ n \leftarrow \mathit{eval}\ x\,;\ \mathit{print}\ n\,;\ \mathit{return}\ (N\ n)\} \\
& \quad \mathit{exec}\ c\ (v : s) \\
\cong & \quad \{\ \text{associativity and left identity monad laws}\ \} \\
& \mathbf{do}\ N\ n \leftarrow \mathit{eval}\ x\,;\ \mathit{print}\ n\,;\ \mathit{exec}\ c\ (N\ n : s) \\
= & \quad \left\{ \begin{array}{l} \text{define: } \mathit{exec}\ (\mathit{PRINT}\ c)\ (N\ n : s) \\ \qquad\qquad = \mathbf{do}\ \mathit{print}\ n\,;\ \mathit{exec}\ c\ (N\ n : s) \end{array} \right\} \\
& \mathbf{do}\ N\ n \leftarrow \mathit{eval}\ x\,;\ \mathit{exec}\ (\mathit{PRINT}\ c)\ (N\ n : s) \\
{}_{\perp}\!\cong & \quad \{\ {}_{\perp}\!\cong\text{-}\mathit{stuck}\ \text{law}\ \} \\
& \mathbf{do}\ v \leftarrow \mathit{eval}\ x\,;\ \mathit{exec}\ (\mathit{PRINT}\ c)\ (v : s) \\
{}_{\perp}\!\cong & \quad \{\ \text{induction hypothesis for } x\ \} \\
& \mathit{exec}\ (\mathit{comp}\ x\ (\mathit{PRINT}\ c))\ s
\end{array}
$$

$$
\begin{array}{ll}
& \mathbf{do}\ v \leftarrow \mathit{eval}\ \mathit{Flip}\,;\ \mathit{exec}\ c\ (v : s) \\
= & \quad \{\ \text{definition of } \mathit{eval}\ \} \\
& \mathbf{do}\ v \leftarrow \mathit{return}\ (B\ \mathit{True}) \oplus \mathit{return}\ (B\ \mathit{False}) \\
& \quad \mathit{exec}\ c\ (v : s) \\
= & \quad \{\ \text{definition of} \gg\ \} \\
& (\mathbf{do}\ v \leftarrow \mathit{return}\ (B\ \mathit{True})\,;\ \mathit{exec}\ c\ (v : s)) \\
& \oplus (\mathbf{do}\ v \leftarrow \mathit{return}\ (B\ \mathit{False})\,;\ \mathit{exec}\ c\ (v : s)) \\
\cong & \quad \{\ \text{left identity monad law}\ \} \\
& \mathit{exec}\ c\ (B\ \mathit{True} : s) \oplus \mathit{exec}\ c\ (B\ \mathit{False} : s) \\
= & \quad \left\{ \begin{array}{l} \text{define: } \mathit{exec}\ (\mathit{FLIP}\ c)\ s = \mathit{exec}\ c\ (B\ \mathit{True} : s) \\ \qquad\qquad\qquad\qquad \oplus\ \mathit{exec}\ c\ (B\ \mathit{False} : s) \end{array} \right\} \\
& \mathit{exec}\ (\mathit{FLIP}\ c)\ s
\end{array}
$$

Fig. 3. Calculation of specification (7) for *Print* and *Flip*.

Target language

$$\mathbf{data}\ \mathit{Code} = \mathit{PUSH}\ \mathit{Value}\ \mathit{Code} \mid \mathit{ADD}\ \mathit{Code} \mid \mathit{PRINT}\ \mathit{Code} \mid \mathit{JPC}\ \mathit{Code}\ \mathit{Code} \mid \mathit{FLIP}\ \mathit{Code} \mid \mathit{HALT}$$

Compiler

$$
\begin{array}{l}
\mathit{compile} :: \mathit{Expr} \to \mathit{Code} \\
\mathit{compile}\ e = \mathit{comp}\ e\ \mathit{HALT} \\
\\
\mathit{comp} :: \mathit{Expr} \to \mathit{Code} \to \mathit{Code} \\
\mathit{comp}\ (\mathit{Val}\ v) \quad c = \mathit{PUSH}\ v\ c \\
\mathit{comp}\ (\mathit{Add}\ x\ y)\ c = \mathit{comp}\ x\ (\mathit{comp}\ y\ (\mathit{ADD}\ c)) \\
\mathit{comp}\ (\mathit{If}\ x\ y\ z)\ c = \mathit{comp}\ x\ (\mathit{JPC}\ (\mathit{comp}\ y\ c)\ (\mathit{comp}\ z\ c)) \\
\mathit{comp}\ (\mathit{Print}\ x) \quad c = \mathit{comp}\ x\ (\mathit{PRINT}\ c) \\
\mathit{comp}\ \mathit{Flip} \quad\quad c = \mathit{FLIP}\ c
\end{array}
$$

Stack machine

$$
\begin{array}{l}
\mathbf{type}\ \mathit{Stack} = [\,\mathit{Value}\,] \\
\\
\mathit{exec} :: \mathit{Code} \to \mathit{Stack} \to \mathit{CTree}_{\perp}\ \mathit{PrintEff}\ \mathit{Stack} \\
\mathit{exec}\ (\mathit{PUSH}\ v\ c)\ s = \mathit{exec}\ c\ (v : s) \\
\mathit{exec}\ (\mathit{ADD}\ c) \quad (N\ n : N\ m : s) \\
\quad = \mathit{exec}\ c\ (N\ (m + n) : s) \\
\mathit{exec}\ (\mathit{JPC}\ c'\ c) \quad (B\ b : s) \\
\quad = \mathbf{if}\ b\ \mathbf{then}\ \mathit{exec}\ c'\ s\ \mathbf{else}\ \mathit{exec}\ c\ s \\
\mathit{exec}\ (\mathit{PRINT}\ c) \quad (N\ n : s) \\
\quad = \mathbf{do}\ \mathit{print}\ n\,;\ \mathit{exec}\ c\ (N\ n : s) \\
\mathit{exec}\ (\mathit{FLIP}\ c) \quad s = \mathit{exec}\ c\ (B\ \mathit{True} : s) \\
\qquad\qquad\qquad\qquad \oplus\ \mathit{exec}\ c\ (B\ \mathit{False} : s) \\
\mathit{exec}\ \mathit{HALT} \quad\quad s = \mathit{return}\ s \\
\mathit{exec}\ \_ \quad\quad\quad\ \ \_ = \mathit{stuck}
\end{array}
$$

Fig. 4. Compiler and stack machine for the simple expression language with print and non-determinism.

As before, we consider the top-level compiler $\mathit{compile} :: \mathit{Expr} \to \mathit{Code}$, whose correctness is captured by the following property:

$$\mathbf{do}\ v \leftarrow \mathit{eval}\ e\,;\ \mathit{return}\ (v : s) \quad {}_{\perp}\!\cong \quad \mathit{exec}\ (\mathit{compile}\ e)\ s \tag{8}$$

The calculation proceeds in the same way as in section 2. The resulting complete definitions of the compiler, target language, and stack machine calculated in this fashion are shown in Figure 4. Similarly to the calculation in section 2, we make the definition of *exec* total by adding a catch-all clause that returns *stuck*, which we are free to choose as the calculation does not depend on it.

The use of the partial correctness property (5) for the compiler calculation in section 2 was merely a convenience, as the calculated compiler in fact satisfies the strict correctness property (1). However, with the addition of effects, a partial correctness property using skew bisimilarity is crucial to permit the compiler to produce more efficient code: Our compiler avoids having to insert *ISN* instructions to check the type of values stored on the stack, which the compiler calculated in section 3.4 had to do to satisfy the stricter specification.

4.2 Bisimilarity vs. Skew Bisimilarity

To illustrate the difference between bisimilarity and skew bisimilarity, let's consider the expression $Add\ (Val\ (B\ True))\ (Print\ (Val\ (N\ 1)))$. Its semantics produces the following transition:

$$eval\ (Add\ (Val\ (B\ True))\ (Print\ (Val\ (N\ 1)))) \xrightarrow{\uparrow Inl\ Stuck} \ldots$$

It gets stuck since it tries to add a Boolean with an integer. In particular, evaluation gets stuck immediately before trying to evaluate the second argument.

The compiler calculated in this section, shown in Figure 4, transforms the above expression into

$$PUSH\ (B\ True)\ (PUSH\ (N\ 1)\ (PRINT\ (ADD\ HALT)))$$

for which we observe the following transitions of the stack machine:

$$\begin{aligned}
&exec\ (PUSH\ (B\ True)\ (PUSH\ (N\ 1)\ (PRINT\ (ADD\ HALT))))\ [\,] \\
&= exec\ (PUSH\ (N\ 1)\ (PRINT\ (ADD\ HALT)))\ [B\ True] \\
&= exec\ (PRINT\ (ADD\ HALT))\ [N\ 1, B\ True] \\
&\xrightarrow{\uparrow Inr\ (PrintInt\ 1)} \lambda() \rightarrow exec\ (ADD\ HALT)\ [N\ 1, B\ True] \\
&\xrightarrow{\downarrow()} exec\ (ADD\ HALT)\ [N\ 1, B\ True] = stuck \\
&\xrightarrow{\uparrow Inl\ Stuck} elimVoid
\end{aligned}$$

That is, while the original expression gets stuck immediately, the compiled code still evaluates the second argument to *Add* and thus issues a print effect before getting stuck. Hence, for this example expression, the strict compiler correctness property (6) does not hold: The semantics of the source expression is *not* bisimilar to the semantics of the corresponding code produced by the compiler.

By contrast, the compiler we would get from the calculation in section 3.4 produces the code

$$PUSH\ (B\ True)\ (ISN\ (PUSH\ (N\ 1)\ (PRINT\ (ADD\ HALT))))$$

for which we observe the following transition:

$$\begin{aligned}
&exec\ (PUSH\ (B\ True)\ (ISN\ (PUSH\ (N\ 1)\ (PRINT\ (ADD\ HALT)))))\ [\,] \\
&= exec\ (ISN\ (PUSH\ (N\ 1)\ (PRINT\ (ADD\ HALT))))\ [B\ True] \\
&= stuck \\
&\xrightarrow{\uparrow Inl\ Stuck} elimVoid
\end{aligned}$$

The additional *ISN* instruction ensures that the machine gets stuck before executing the code $PUSH\ (N\ 1)\ (PRINT\ (ADD\ HALT))$ that would otherwise produce an observable effect.

4.3 Type Safety

The compiler specification (8) only captures partial correctness. However, we can strengthen this partial correctness property to a strict specification if we restrict ourselves to well-typed expressions.

$$\frac{P(v)}{\mathit{return}\ v \text{ is } P\text{-safe}} \qquad \frac{}{\mathit{Zero} \text{ is } P\text{-safe}} \qquad \frac{p \text{ and } q \text{ are } P\text{-safe}}{p \oplus q \text{ is } P\text{-safe}} \qquad \frac{p \text{ is } P\text{-safe}}{\mathit{Later}\ p \text{ is } P\text{-safe}}$$

$$\frac{c\ v \text{ is } P\text{-safe for all } v}{\mathit{Eff}\ (\mathit{Inr}\ o)\ c \text{ is } P\text{-safe}} \qquad \frac{p \text{ is } P\text{-safe} \qquad f\ v \text{ is } Q\text{-safe for all } v \text{ with } P(v)}{p \gg\!= f \text{ is } Q\text{-safe}}$$

Fig. 5. Proof rules for P-safety.

To this end, we consider a simple type system for our source language:

$$\frac{}{\vdash_{\mathsf{v}} B\ b : \mathit{Bool}} \qquad \frac{}{\vdash_{\mathsf{v}} N\ n : \mathit{Int}} \qquad \frac{\vdash_{\mathsf{v}} v : \tau}{\vdash \mathit{Val}\ v : \tau} \qquad \frac{\vdash e_1 : \mathit{Int} \qquad \vdash e_2 : \mathit{Int}}{\vdash \mathit{Add}\ e_1\ e_2 : \mathit{Int}}$$

$$\frac{\vdash e_1 : \mathit{Bool} \qquad \vdash e_2 : \tau \qquad \vdash e_3 : \tau}{\vdash \mathit{If}\ e_1\ e_2\ e_3 : \tau} \qquad \frac{\vdash e : \mathit{Int}}{\vdash \mathit{Print}\ e : \mathit{Int}} \qquad \frac{}{\vdash \mathit{Flip} : \mathit{Bool}}$$

Our goal is to invoke Proposition 1 to strengthen the compiler correctness property (8) to

$$\mathbf{do}\ v \leftarrow \mathit{eval}\ e;\ \mathit{return}\ (v : s) \quad \cong \quad \mathit{exec}\ (\mathit{compile}\ e)\ s \qquad \text{if } \vdash e : \tau \tag{9}$$

What remains to be shown, so that we can apply Proposition 1, is that the left-hand side of the above equation is a safe choice tree, which in turn requires us to show that

$$\mathit{eval}\ e \text{ is safe for all } \vdash e : \tau \tag{10}$$

To prove this by induction on e, we need to strengthen the property so that it additionally states that $\mathit{eval}\ e$ only produces values of type τ. To this end, we generalise the safety predicate to P-safety, where P is a predicate on the values of type a produced by a choice tree of type $\mathit{CTree}\ e\ a$. We define *P-safety* as the largest predicate with the following property:

If p is P-safe, then p is locally safe, any q with $p \overset{l}{\Longrightarrow} q$ is P-safe, and $P(v)$ whenever $p \overset{\langle v \rangle}{\Longrightarrow} q$.

We thus strengthen (10) to the following property:

$$\mathit{eval}\ e \text{ is } P_\tau\text{-safe for all } \vdash e : \tau, \qquad \text{where } P_\tau(v) \text{ iff } \vdash_{\mathsf{v}} v : \tau \tag{11}$$

To prove this, we may use the proof rules shown in Figure 5. The rule for *Later* is denoted with a double line to indicate that it is a coinductive rule. The proof of (11) proceeds by a straightforward induction on the derivation of $\vdash e : \tau$. For example, in the case for $\vdash \mathit{If}\ x\ y\ z : \tau$, we have by induction hypothesis that $\mathit{eval}\ x$ is P_{Bool}-safe. Hence, by the proof rule for $\gg\!=$, it remains to be shown that $\mathbf{if}\ b\ \mathbf{then}\ \mathit{eval}\ y\ \mathbf{else}\ \mathit{eval}\ z$ is P_τ-safe for all $b :: \mathit{Bool}$. For that it suffices to show that $\mathit{eval}\ y$ and $\mathit{eval}\ z$ are P_τ-safe, which follows from the induction hypothesis.

That means, by (11) and Proposition 1, we have the compiler correctness property (9).

4.4 Extension to Non-termination and Concurrency

We demonstrated the compiler calculation technique based on skew bisimilarity on a very simple source language. However, this technique also applies to much more feature-rich languages. We only sketch two such examples here and give the derived definitions in Appendices A and B. The full details of the calculations can be found in the accompanying Agda formalisation [Bahr 2026].

4.4.1 Non-termination. To reason about non-terminating languages, Bahr and Hutton [2022] use a step-indexed version of bisimilarity, denoted $\cong_i$, so that the calculation can proceed by induction on both the structure of the source expression e and the step index i. This allows the calculation to use the induction hypothesis under occurrences of *Later* by using the following congruence rule:

$$\mathit{Later}\ p \ \cong_i\ \mathit{Later}\ q \qquad \text{if} \quad p \cong_j q \text{ for all } j < i$$

That is, whenever we want to prove step-indexed bisimilarity at i, we only have to prove step-indexed bisimilarity at all j smaller than i, for which we can then apply the induction hypothesis.

We can also formulate skew bisimilarity in a step-indexed version, denoted $\mathrel{\perp}\cong_i$. While $\mathrel{\perp}\cong$ is defined coinductively, $\mathrel{\perp}\cong_i$ is defined inductively as the *smallest* relation such that $p \mathrel{\perp}\cong_0 q$ holds, and $p \mathrel{\perp}\cong_{i+1} q$ holds whenever the following two conditions hold when p is locally safe:

(1) If $p \overset{l}{\Longrightarrow} p'$ then there is some $q \overset{l}{\Longrightarrow} q'$ with $p' \mathrel{\perp}\cong_j q'$
(2) If $q \overset{l}{\Longrightarrow} q'$ then there is some $p \overset{l}{\Longrightarrow} p'$ with $p' \mathrel{\perp}\cong_j q'$

$$\text{where } j = \begin{cases} i & \text{if } l = \tau \\ i+1 & \text{otherwise} \end{cases}$$

This step-indexed version of skew bisimilarity approximates skew bisimilarity, i.e. $p \mathrel{\perp}\cong q$ iff $p \mathrel{\perp}\cong_i q$ for all i. Moreover, it satisfies the same congruence laws as step-indexed bisimilarity (cf. Bahr and Hutton [2023]). In particular, we have the following congruence law for *Later* that allows us to reason about non-terminating languages:

$$\mathit{Later}\ p \ \mathrel{\perp}\cong_i\ \mathit{Later}\ q \qquad \text{if} \quad p \mathrel{\perp}\cong_j q \text{ for all } j < i \qquad (\mathrel{\perp}\cong_i\text{-}\mathit{Later})$$

We have applied step-indexed skew bisimilarity to calculate compilers for various lambda calculi. In particular, we have calculated a compiler for Rattus [Bahr 2022], a lambda calculus with a sophisticated Fitch-style type system [Clouston 2018] featuring LTL-inspired modal type operators and guarded recursion. The complete definitions of the source language semantics, the stack machine, and the compiler are found in Appendix A. Bahr [2022] gives the semantics as a big-step relation $(e, \sigma) \Downarrow (v, \sigma')$ that evaluates an expression e in the context of a store σ to a value v and a new store σ'. Following Bahr and Hutton [2022], we translated this semantics in the standard way into the form $(e, \sigma) \Downarrow_\gamma (v, \sigma')$ that uses an environment γ rather than substitutions. We then turned this semantics into an evaluation function. For example, the two big-step semantics rules for application and the advance operator

$$\frac{(t_1, \sigma) \Downarrow_\gamma (\langle x, t_1', \gamma' \rangle, \sigma_1) \quad (t_2, \sigma_1) \Downarrow_\gamma (v_2, \sigma_2) \quad (t_1', \sigma_2) \Downarrow_{\gamma'[x \mapsto v_2]} (v, \sigma_3)}{(t_1\ t_2, \sigma) \Downarrow_\gamma (v, \sigma_3)} \qquad \frac{(t, \eta_N) \Downarrow_\gamma (l, \eta_N') \quad \eta_N'(l) = \langle t', \gamma' \rangle \quad (t', \eta_N' \checkmark \eta_L) \Downarrow_{\gamma'} (v, \sigma)}{(\mathrm{adv}\, t, \eta_N \checkmark \eta_L) \Downarrow_\gamma (v, \sigma)}$$

are translated into the following clauses for the evaluation function *eval*:

$$\begin{aligned} &\mathit{eval}\ (\mathit{App}\ t_1\ t_2)\ \gamma\ \sigma = \\ &\quad \mathbf{do}\ (\mathit{Clo}_V\ t_1'\ \gamma', \sigma_1) \leftarrow \mathit{eval}\ t_1\ \gamma\ \sigma \\ &\qquad (v_2, \sigma_2) \qquad\quad \leftarrow \mathit{eval}\ t_2\ \gamma\ \sigma_1 \\ &\qquad \mathit{Later}\ (\mathit{eval}\ t_1'\ (v_2 : \gamma')\ \sigma_2) \end{aligned} \qquad \begin{aligned} &\mathit{eval}\ (\mathit{Adv}\ t)\ \gamma\ (\eta_N \checkmark \eta_L) = \\ &\quad \mathbf{do}\ (\mathit{Loc}_V\ l, N\ \eta_N') \leftarrow \mathit{eval}\ t\ \gamma\ (N\ \eta_N) \\ &\qquad (t', \gamma') \qquad\quad \leftarrow \mathit{lookup}\ l\ \eta_N' \\ &\qquad \mathit{Later}\ (\mathit{eval}\ t'\ \gamma'\ (\eta_N' \checkmark \eta_L)) \end{aligned}$$

Note that the recursive calls of *eval* on t_1' and t' are not structurally recursive and to ensure that *eval* is well-defined, these recursive calls are guarded by *Later*. In addition, this allows the calculation to use $\mathrel{\perp}\cong_i$-*Later* to apply the induction hypotheses for t_1' and t' at a smaller step index $j < i$. The *Adv* operator turns the store from a two-heap store $\eta_N \checkmark \eta_L$ into a single-heap store $N\ \eta_N$ before evaluating its argument t and subsequently recovering the second heap η_L. Finally, we use de Bruijn indices to represent variables, so that the free variable x in the closure $\langle x, t_1', \gamma' \rangle$ is implicitly represented by the de Bruijn index 0, and the environment γ' is extended by the mapping $x \mapsto v_2$ by $v_2 : \gamma'$ since environments are represented as lists of values.

The stack machine uses configurations of the form (s, γ, σ) consisting of a stack, an environment, and a store. To see how this works, we focus on how the compiler translates the advance operator:

$$comp\ (Adv\ x)\ c = ADV\ (comp\ x\ (ENDADV\ c))$$

To match the semantics of the *Adv* operator, the *ADV* instruction saves the heap η_L on the stack before executing the code for x, while the *ENDADV* instruction recovers η_L from the stack and initiates the execution of the delayed computation from the store:

$$\begin{aligned}
&exec\ (ADV\ c)\ (s, \gamma, \eta_N \checkmark \eta_L) = exec\ c\ (HEAP\ \eta_L : s, \gamma, N\ \eta_N)\\
&exec\ (ENDADV\ c)\ (VAL\ (Loc_V^M\ l) : HEAP\ \eta_L : s, \gamma, N\ \eta_N) =\\
&\quad \mathbf{do}\ (c', \gamma') \leftarrow lookup\ l\ \eta_N\ ;\ Later\ (exec\ c'\ (CLO\ c\ \gamma : s, \gamma', \eta_N \checkmark \eta_L))
\end{aligned}$$

4.4.2 Concurrency. Choice trees can be used to give semantics to programming languages with concurrency features [Chappe et al. 2025]. For example, Bahr and Hutton [2023] have used choice trees to calculate compilers for concurrent languages including a concurrent call-by-value lambda calculus. However, their calculation technique requires the construction of *codensity choice trees*, which wrap choice trees inside the *codensity monad* [Voigtländer 2008]:

$$\begin{aligned}
&\mathbf{type}\ CTree^{\mathrm{c}}\ \ e\ a = forall\ r\ .\ (a \longrightarrow CTree\ e\ r) \longrightarrow CTree\ e\ r\\
&\mathbf{type}\ CTree^{\mathrm{c}}_{\perp}\ e\ a = CTree^{\mathrm{c}}\ (Stuck \uplus e)\ a
\end{aligned}$$

As Bahr and Hutton [2023] show, the operations on choice trees (*Eff*, $\oplus$, *Zero* etc.) can be lifted to corresponding operations on codensity choice trees (Eff_{c}, $\oplus_{\mathrm{c}}$, $Zero_{\mathrm{c}}$ etc.), and the type $CTree^{\mathrm{c}}$ has an additional parallel composition operator $\vec{\shortparallel}_{\mathrm{c}}$. We then lift the (step-indexed) bisimilarity relation to codensity choice trees by applying them to the trivial continuation *return*, i.e. defining $p \cong_i q$ iff $p\ return \cong_i q\ return$. Following this idea, we lift the notion of (step-indexed) skew bisimilarity to partial codensity trees $p, q :: CTree^{\mathrm{c}}_{\perp}\ e\ a$ by defining $p\ {}_{\perp}\!\cong_i q$ iff $p\ return\ {}_{\perp}\!\cong_i q\ return$.

We evaluated step-indexed skew bisimilarity on codensity choice trees by calculating a compiler for the concurrent lambda calculus with channel-based communication, for which Bahr and Hutton [2023] have previously calculated a compiler. This language is a lambda calculus extended with integers; an operation $\mathsf{send}(s, t)$ that sends t on channel s; an operation $\mathsf{receive}(t)$ that receives a value from channel t; and an operation $\mathsf{fork}(x.t)$ that allocates a new channel c, binds c to x, and evaluates t in parallel. The semantics of this language uses the parallel composition operator $\vec{\shortparallel}_{\mathrm{c}}$ as well as an algebraic effect with three operations to create channels, send data on channels, and receive data from channels. Our calculation is similar to Bahr and Hutton's, but with three key differences: First, we use the type $CTree^{\mathrm{c}}_{\perp}$ instead of $CTree^{\mathrm{c}}$ so that we can represent unsafe behaviour using $stuck_{\mathrm{c}}$ rather than $Zero_{\mathrm{c}}$, where $stuck_{\mathrm{c}} = Eff_{\mathrm{c}}\ (Inl\ Stuck)\ elimVoid$. Second, our calculation uses the law $stuck_{\mathrm{c}}\ {}_{\perp}\!\cong_i p$ to safely ignore unsafe behaviour. Third, by using this law for $stuck_{\mathrm{c}}$, we simplify the calculation and avoid additional instructions analogous to the *ISN* instruction from section 3.4. For example, our compiler translates applications as follows:

$$comp\ (App\ x\ y)\ c = comp\ x\ (comp\ y\ (APP\ c))$$

By contrast, Bahr and Hutton's compiler produces $comp\ x\ (ISCLO\ (comp\ y\ (APP\ c)))$, where *ISCLO* checks whether the top of the stack contains a closure. Our calculation avoids the overhead of these instructions while still obtaining the full correctness property for well-typed terms, which we prove using a corresponding version of Proposition 1 for codensity choice trees.

5 Ordered Skew Bisimilarity for Register Machines

The use of skew bisimilarity enables compiler calculations to gracefully deal with unsafe behaviours in the source language semantics. In this section, we demonstrate that with a mild extension, skew

bisimilarity may also be used to enable a novel calculation technique for deriving compilers for *register machines*, which store data in named memory cells called registers.

We expect a compiler for a register machine to produce code that simply overwrites unused registers rather than explicitly freeing them. This aspect of register machines complicates reasoning significantly. But a small trick, proposed by Bahr and Hutton [2020], can simplify reasoning: The target machine's memory is endowed with a partial order $\sqsubseteq$ so that two memory instances m and m' are ordered $m' \sqsubseteq m$ iff we can obtain m' from m by freeing some registers. Importantly, such freeing of registers only happens at the meta level, i.e. during the calculation, and thus no explicit freeing of registers is necessary during runtime. However, freeing of registers may introduce unsafe behaviour, since we might free a register from which we later try to read. That is why we need to be able to reason about unsafe behaviours even in the context of source languages with no such unsafe behaviours. So far, Bahr and Hutton's calculation technique for register machines has only been applied to pure languages and only partially to non-terminating languages. By combining this technique with (codensity) choice trees and skew bisimilarity, we can extend it to a much wider range of languages and language features.

5.1 Reasoning About Register Machines

We begin by recalling the memory model of Bahr and Hutton [2020], rephrased slightly to match our semantic framework: It consists of an abstract type *Reg* of registers, a type *Mem a* of memory instances that map registers to values of type *a*, and the following set of operations:

$$\begin{array}{lll} \mathit{first} :: \mathit{Reg} & \mathit{empty} :: \mathit{Mem}\ a & \mathit{set} :: \mathit{Reg} \to a \to \mathit{Mem}\ a \to \mathit{Mem}\ a \\ \mathit{next} :: \mathit{Reg} \to \mathit{Reg} & & \mathit{get} :: \mathit{Reg} \to \mathit{Mem}\ a \to \mathit{CTree}_{\perp}\ e\ a \end{array}$$

Intuitively, *first* is the first register, and *next* produces a fresh register given an existing register. That is, we assume an unbounded set of registers. In turn, *empty* produces an empty memory instance, *set* writes a value in the given register, and *get* reads the value of a given register.

This abstract model comes with a set of laws that specify the semantics of the memory operations. To formulate these laws, the model uses a partial order $\sqsubseteq$ on *Mem a* and a binary predicate *freeFrom*. As mentioned above, $m' \sqsubseteq m$ means that we can obtain m' from m by freeing some registers, whereas *freeFrom r m* means that register *r* and all registers following *r* are free in memory *m*:

$$\begin{array}{rcll} \mathit{freeFrom}\ \mathit{first}\ \mathit{empty} & & & (\mathit{first}\text{-}\mathit{empty}) \\ \mathit{get}\ r\ (\mathit{set}\ r\ v\ m) & = & \mathit{return}\ v & (\mathit{set}\text{-}\mathit{get}) \\ \mathit{freeFrom}\ r\ m & \Rightarrow & m \sqsubseteq \mathit{set}\ r\ v\ m & (\mathit{freeFrom}\text{-}\mathit{set}) \\ \mathit{freeFrom}\ r\ m & \Rightarrow & \mathit{freeFrom}\ (\mathit{next}\ r)\ (\mathit{set}\ r\ v\ m) & (\mathit{freeFrom}\text{-}\mathit{next}) \\ m \sqsubseteq m' & \Rightarrow & \mathit{set}\ r\ v\ m \sqsubseteq \mathit{set}\ r\ v\ m' & (\mathit{set}\text{-monotone}) \\ m \sqsubseteq m' & \Rightarrow & \mathit{get}\ r\ m \mathrel{{}_{\perp}\!\cong} \mathit{get}\ r\ m' & (\mathit{get}\text{-monotone}) \end{array}$$

We deviate slightly from the original presentation of the memory model of Bahr and Hutton [2020]. In their presentation, *get* has type $\mathit{Reg} \to \mathit{Mem}\ a \to a$ since they use a non-total metalanguage. By contrast, the approach in this article is to use a total metalanguage and to reason explicitly about partiality using partial choice trees instead. Consequently, the *set-get* and *get*-monotone laws also deviate slightly from the original presentation. Our revised model can easily be shown to be consistent by defining $\mathit{Reg} = \mathit{Int}$ and $\mathit{Mem}\ a = \mathit{Reg} \to \mathit{Maybe}\ a$.

Bahr and Hutton [2020] consider source languages with a pure semantics of the form $\mathit{eval} :: \mathit{Expr} \to \mathit{Value}$ and use the above memory model to calculate compilers that target register machines of the form $\mathit{exec} :: \mathit{Code} \to \mathit{Conf} \to \mathit{Conf}$, where *Conf* represents the machine's configurations:

type $Conf = (Acc, Mem\ Value)$ **type** $Acc = Value$

That is, configurations are pairs (a, m) consisting of a distinguished accumulation register a and a memory m with further registers. To formulate the compiler correctness property, Bahr and Hutton extend the partial order $\sqsubseteq$ on Mem to machine configurations:

$$(a, m) \sqsubseteq (a', m') \iff a = a' \ \wedge \ m \sqsubseteq m'$$

In this pure setting, the compiler correctness property is then phrased as follows:

$$(eval\ e, empty) \sqsubseteq exec\ (compile\ e)\ (a, empty) \tag{12}$$

The use of the partial order $\sqsubseteq$ means that we allow the machine configuration on the right-hand side to have stored values in additional registers. In other words, the code produced by the compiler may use free registers to store intermediate values without having to explicitly free them afterwards. In addition, by leaving the value a fully unconstrained, the specification expresses the fact that the code for e may freely overwrite the accumulation register.

As Bahr and Hutton [2020] have demonstrated, this setup allows us to calculate a compiler targeting a register machine. However, this approach is limited to terminating languages without any side effects. While Bahr and Hutton have also applied this technique to non-terminating languages such as the untyped lambda calculus, their compiler specification only captures completeness, i.e. all behaviours of the source program are also exhibited by the target code, but not soundness, i.e. the property that all behaviours of the target code are also exhibited by the original source program.

5.2 Ordered Skew Bisimilarity

To combine the compiler calculation technique sketched above with choice trees, and thereby accommodate effects, we generalise the skew bisimilarity relation $\mathrel{\perp}\!\cong$ so that it also incorporates the order $\sqsubseteq$ on machine configurations. To define such a relation on partial choice trees of type $CTree_\perp\ e\ a$, we assume an order $\sqsubseteq$ on a and extend it to the set of labels in the labelled transition system. The order $\sqsubseteq$ on labels is the least reflexive relation such that $\langle v \rangle \sqsubseteq \langle v' \rangle$ whenever $v \sqsubseteq v'$. Recall that $\langle v \rangle$ is the transition label produced by the choice tree $return\ v$. We then define the *ordered skew bisimilarity* relation, denoted $\mathrel{\perp}\!\lesssim$, as the largest relation that satisfies the following:

(1) If $p \mathrel{\perp}\!\lesssim q$, p is locally safe, and $p \overset{l}{\Longrightarrow} p'$, then there is some $q \overset{l'}{\Longrightarrow} q'$ with $p' \mathrel{\perp}\!\lesssim q'$, and $l \sqsubseteq l'$.
(2) If $p \mathrel{\perp}\!\lesssim q$, p is locally safe, and $q \overset{l}{\Longrightarrow} q'$, then there is some $p \overset{l'}{\Longrightarrow} p'$ with $p' \mathrel{\perp}\!\lesssim q'$, and $l' \sqsubseteq l$.

In contrast to skew bisimilarity, ordered skew bisimilarity does not require that labels of the form $\langle v \rangle$ match on the nose, but instead a label $\langle v \rangle$ on the left-hand side has to match with a label $\langle v' \rangle$ on the right-hand side such that $v \sqsubseteq v'$ and vice versa. For example, $return\ v \mathrel{\perp}\!\lesssim return\ v'$ holds whenever $v \sqsubseteq v'$. With the help of ordered skew bisimilarity, we can now formulate the compiler correctness property for register machines within the semantic framework of choice trees.

To reason about ordered skew bisimilarity, we have congruence laws at our disposal that are similar to those for skew bisimilarity, but to account for the underlying ordering $\sqsubseteq$, functions must be monotone ($f\ x \mathrel{\perp}\!\lesssim g\ y$ whenever $x \sqsubseteq y$) rather than just ordered pointwise as for skew bisimilarity ($f\ x \mathrel{\perp}\!\cong g\ x$ for all x). For example, the congruence law for $\gg\!=$ now reads as follows:

$$p \gg\!= f \ \mathrel{\perp}\!\lesssim \ q \gg\!= g \qquad \text{if } p \mathrel{\perp}\!\lesssim q \quad \text{and} \quad f\ x \mathrel{\perp}\!\lesssim g\ y \ \text{ for all } x, y \text{ with } x \sqsubseteq y \quad (\gg\!=\text{-congruence})$$

In addition, we have the following relation between skew bisimilarity and ordered skew bisimilarity:

Proposition 2. *Let $p, q :: CTree_\perp\ e\ a$ be two partial choice trees.*
(i) If $p \mathrel{\perp}\!\cong q$, then $p \mathrel{\perp}\!\lesssim q$.
(ii) If $x \sqsubseteq y$ implies $x = y$ for all $x, y :: a$, then $p \mathrel{\perp}\!\lesssim q$ implies $p \mathrel{\perp}\!\cong q$.

As a special case, we obtain the following corollary, which states that, for safe choice trees, ordered skew bisimilarity implies full bisimilarity as long as we disregard the final return value:

Corollary 3. *Let $p, q :: CTree_\perp\ e\ a$, $f :: a \to b$ monotone, and $\sqsubseteq$ on b such that $x \sqsubseteq y$ implies $x = y$ for all $x, y :: b$. If p is safe and $p \mathrel{\perp\lesssim} q$, then $fmap\ f\ p \cong fmap\ f\ q$.*

Similar to our use of Proposition 1 for compiler calculations based on skew bisimilarity, we can use this corollary to derive the full correctness property for safe source programs.

5.3 Calculating a Register Machine Compiler

To demonstrate the calculation approach for register machines, we reconsider the expression language from section 3, but without conditionals and non-determinism as these are not necessary here to illustrate the calculation technique:

$$\textbf{data}\ Expr = Val\ Int \mid Add\ Expr\ Expr \mid Print\ Expr$$

$$\begin{aligned}
&eval :: Expr \to CTree_\perp\ PrintEff\ Int\\
&eval\ (Val\ n) \quad\ = return\ n\\
&eval\ (Add\ x\ y) = \textbf{do}\ m \leftarrow eval\ x\,;\ n \leftarrow eval\ y\,;\ return\ (m + n)\\
&eval\ (Print\ x) \ = \textbf{do}\ n \leftarrow eval\ x\,;\ print\ n\,;\ return\ n
\end{aligned}$$

We have specified the semantics using partial choice trees, although the semantics never gets stuck, i.e. we can prove that *eval e* is safe for all $e :: Expr$. We use the type $CTree_\perp\ PrintEff\ Int$ instead of just $CTree\ PrintEff\ Int$, since (ordered) skew bisimilarity is only defined on partial choice trees.

We aim to calculate a compiler $compile :: Expr \to Code$ with a target language *Code* that runs on a register machine $exec :: Code \to Conf \to CTree_\perp\ PrintEff\ Conf$ where

$$\textbf{type}\ Conf = (Acc, Mem\ Int) \qquad\qquad \textbf{type}\ Acc = Int$$

The compiler correctness property is similar to specification (12) but is now formulated using partial choice trees and ordered skew bisimilarity instead:

$$\textbf{do}\ v \leftarrow eval\ e\,;\ return\ (v, empty) \mathrel{\perp\lesssim} exec\ (compile\ e)\ (a, empty) \tag{13}$$

In order to calculate the compiler, we follow the approach of Bahr and Hutton [2020] and generalise its type to $comp :: Expr \to Reg \to Code \to Code$ so that it takes two additional arguments: a code continuation *c*, similarly to stack machine compilers, and a register *r*, which indicates the first register that the compiler is at liberty to use. To accommodate this generalisation, the specification has to be generalised accordingly. In addition to the code continuation *c*, we generalise the specification so that the machine may start with any memory *m* (not just the empty memory) as long as *m* is free from the register *r* onwards:

$$freeFrom\ r\ m \quad \Rightarrow \quad \textbf{do}\ v \leftarrow eval\ e\,;\ exec\ c\ (v, m) \mathrel{\perp\lesssim} exec\ (comp\ e\ r\ c)\ (a, m) \tag{14}$$

The final crucial component of the calculation technique of Bahr and Hutton [2020] is a side condition that requires the machine *exec* to be monotone with respect to the partial order $\sqsubseteq$ on machine configurations. In our semantic framework of choice trees, this condition reads as follows:

$$s \sqsubseteq s' \ \Rightarrow\ exec\ c\ s \mathrel{\perp\lesssim} exec\ c\ s' \qquad (exec\text{-monotone})$$

Intuitively speaking, the monotonicity property for *exec* means that the machine cannot observe whether reading a register has failed or not. This property holds by construction, since the two operations on memory configurations – *set* and *get* – are monotone. As we shall see, the compiler calculation crucially relies on the *exec*-monotone property.

We now calculate the definition of *comp* from specification (14) by structural induction on *e*. That is, we may assume some memory *m* and register *r* with *freeFrom r m* and then aim to manipulate

the left-hand side of the inequality into a term of the form $exec\ c'\ (a, m)$, which can be made to match the right-hand side of the inequality by defining $comp\ e\ r\ c = c'$ for that case of e.

The case for $e = Val\ n$ is similar to the calculation for a stack machine in section 3:

$$
\begin{aligned}
& \mathbf{do}\ v \leftarrow eval\ (Val\ n)\,;\ exec\ c\ (v, m) \\
= \quad & \{\,\text{definition of } eval\,\} \\
& \mathbf{do}\ v \leftarrow return\ n\,;\ exec\ c\ (v, m) \\
\cong \quad & \{\,\text{left identity monad law}\,\} \\
& exec\ c\ (n, m) \\
= \quad & \{\,\text{define: } exec\ (CONST\ n\ c)\ (a, m) = exec\ c\ (n, m)\,\} \\
& exec\ (CONST\ n\ c)\ (a, m)
\end{aligned}
$$

In the last step, similarly to the calculation for a stack machine, we aim to arrive at a term of the form $exec\ c'\ (a, m)$. That is, we must solve the inequality $exec\ c'\ (a, m) \gtrsim_\bot exec\ c\ (n, m)$ for some c', where we write $\gtrsim_\bot$ for the inverse relation of $_\bot\lesssim$. We solve this inequality by first strengthening it to an equation $exec\ c'\ (a, m) = exec\ c\ (n, m)$ and then solving the equation by instantiating c' with $CONST\ n\ c$ so that we can take the equation as a definitional clause for $exec$.

We proceed with the case for $e = Add\ x\ y$, which starts with the typical steps of first applying the definition of $eval$ followed by applying the monad laws:

$$
\begin{aligned}
& \mathbf{do}\ v \leftarrow eval\ (Add\ x\ y)\,;\ exec\ c\ (v, m) \\
= \quad & \{\,\text{definition of } eval\,\} \\
& \mathbf{do}\ v \leftarrow \mathbf{do}\ \{\,n_1 \leftarrow eval\ x\,;\ n_2 \leftarrow eval\ y\,;\ return\ (n_1 + n_2)\,\}\,;\ exec\ c\ (v, m) \\
\cong \quad & \{\,\text{associativity and left identity monad laws}\,\} \\
& \mathbf{do}\ n_1 \leftarrow eval\ x\,;\ n_2 \leftarrow eval\ y\,;\ exec\ c\ (n_1 + n_2, m)
\end{aligned}
$$

We now aim to manipulate this term so that we may apply the induction hypothesis. That is, we must obtain a subterm of the form $\mathbf{do}\ n_2 \leftarrow eval\ y\,;\ exec\ c'\ (n_2, m')$. Given the term we currently have in our calculation, this means that we must solve the inequality

$$\mathbf{do}\ n_2 \leftarrow eval\ y\,;\ exec\ c'\ (n_2, m') \gtrsim_\bot \mathbf{do}\ n_2 \leftarrow eval\ y\,;\ exec\ c\ (n_1 + n_2, m) \qquad \text{for some } c', m'$$

Recall that in the calculation for stack machines, solving this kind of inequality was achieved by introducing an instruction that expects the two operands n_1 and n_2 to appear on the top of the stack and replaces them with their sum $n_1 + n_2$. The same idea works here as well, but instead of using the stack, we must manipulate the memory m. In our target term on the left-hand side, n_2 is already in the accumulation register. It thus remains to pick a suitable m' that contains the other operand n_1. We can achieve this by setting $m' = set\ r\ n_1\ m$. The memory laws allow us to make this transformation since we know that $freeFrom\ r\ m$ holds:

$$
\begin{aligned}
& \mathbf{do}\ n_2 \leftarrow eval\ y\,;\ exec\ c\ (n_1 + n_2, m) \\
{}_\bot\lesssim \quad & \{\,exec\text{-monotone and } freeFrom\text{-}set\,\} \\
& \mathbf{do}\ n_2 \leftarrow eval\ y\,;\ exec\ c\ (n_1 + n_2, set\ r\ n_1\ m)
\end{aligned}
$$

For the induction hypothesis to be applicable, it thus remains to solve the inequality:

$$exec\ c'\ (n_2, m') \gtrsim_\bot exec\ c\ (n_1 + n_2, set\ r\ n_1\ m) \qquad \text{for some } c', m'$$

To this end, we first strengthen this inequality to an equation (i.e. replace $\gtrsim_\bot$ with $=$) and then instantiate $m' = set\ r\ n_1\ m$ and $c' = ADD\ r\ c$:

$$exec\ (ADD\ r\ c)\ (n_2, set\ r\ n_1\ m) = exec\ c\ (n_1 + n_2, set\ r\ n_1\ m)$$

We cannot yet take this equation as a clause for the definition of *exec* since $set\ r\ n_1\ m$ is not a pattern. To rectify this, we first rewrite the right-hand side into an equivalent form by appealing to the *set-get* law and the monad laws:

$$exec\ (ADD\ r\ c)\ (n_2, set\ r\ n_1\ m) \cong \mathbf{do}\ n_1 \leftarrow get\ r\ (set\ r\ n_1\ m)\,;\ exec\ c\ (n_1 + n_2, set\ r\ n_1\ m)$$

Finally, we can read this equation as a definition by generalising $set\ r\ n_1\ m$ to any memory m':

$$exec\ (ADD\ r\ c)\ (n_2, m') = \mathbf{do}\ n_1 \leftarrow get\ r\ m'\,;\ exec\ c\ (n_1 + n_2, m')$$

Using these ideas, we resume our calculation by first manipulating the memory, then introducing the new instruction, and finally applying the induction hypothesis:

$$\begin{array}{ll}
& \mathbf{do}\ n_1 \leftarrow eval\ x\,;\ n_2 \leftarrow eval\ y\,;\ exec\ c\ (n_1 + n_2, m) \\
\lesssim_\perp & \quad \{\ exec\text{-monotone and } freeFrom\text{-}set\ \} \\
& \mathbf{do}\ n_1 \leftarrow eval\ x\,;\ n_2 \leftarrow eval\ y\,;\ exec\ c\ (n_1 + n_2, set\ r\ n_1\ m) \\
\cong & \quad \{\ set\text{-}get\text{; monad laws; define: } exec\ (ADD\ r\ c)\ (a, m) = \mathbf{do}\ b \leftarrow get\ r\ m\,;\ exec\ c\ (b + a, m)\ \} \\
& \mathbf{do}\ n_1 \leftarrow eval\ x\,;\ n_2 \leftarrow eval\ y\,;\ exec\ (ADD\ r\ c)\ (n_2, set\ r\ n_1\ m) \\
\lesssim_\perp & \quad \{\ \text{induction hypothesis for } y \text{ and } next\ r\text{; } freeFrom\text{-}next \text{ law}\ \} \\
& \mathbf{do}\ n_1 \leftarrow eval\ x\,;\ exec\ (comp\ y\ (next\ r)\ (ADD\ r\ c))\ (n_1, set\ r\ n_1\ m)
\end{array}$$

In the last step, we cannot apply the induction hypothesis for the register r since this register is no longer free, i.e. $freeFrom\ r\ (set\ r\ n_1\ m)$ does not hold. Instead, we can apply the induction hypothesis for the next register $next\ r$ since $freeFrom\ (next\ r)\ (set\ r\ n_1\ m)$ according to the *freeFrom-next* law.

After clearing this hurdle, the calculation continues to the final goal in the now familiar fashion. In order to apply the induction hypothesis for x, we need to solve an inequality, which we do by introducing a new instruction and a corresponding definitional clause for *exec*:

$$\begin{array}{ll}
& \mathbf{do}\ n_1 \leftarrow eval\ x\,;\ exec\ (comp\ y\ (next\ r)\ (ADD\ r\ c))\ (n_1, set\ r\ n_1\ m) \\
= & \quad \{\ \text{define: } exec\ (STORE\ r\ c)\ (a, m) = exec\ c\ (a, set\ r\ a\ m)\ \} \\
& \mathbf{do}\ n_1 \leftarrow eval\ x\,;\ exec\ (STORE\ r\ (comp\ y\ (next\ r)\ (ADD\ r\ c)))\ (n_1, m) \\
\lesssim_\perp & \quad \{\ \text{induction hypothesis for } x \text{ and } r\ \} \\
& exec\ (comp\ x\ r\ (STORE\ r\ (comp\ y\ (next\ r)\ (ADD\ r\ c))))\ (a, m)
\end{array}$$

Finally, we conclude the compiler calculation with the case for $e = Print\ x$. We first apply the definition of *eval* and the monad laws. Then, our goal is to manipulate the term so that the induction hypothesis applies. Solving the corresponding inequality is achieved by introducing an instruction:

$$\begin{array}{ll}
& \mathbf{do}\ v \leftarrow eval\ (Print\ x)\,;\ exec\ c\ (v, m) \\
= & \quad \{\ \text{definition of } eval\ \} \\
& \mathbf{do}\ v \leftarrow \mathbf{do}\ \{n \leftarrow eval\ x\,;\ print\ n\,;\ return\ n\}\,;\ exec\ c\ (v, m) \\
\cong & \quad \{\ \text{monad laws}\ \} \\
& \mathbf{do}\ n \leftarrow eval\ x\,;\ print\ n\,;\ exec\ c\ (n, m) \\
= & \quad \{\ \text{define: } exec\ (PRINT\ c)\ (n, m) = \mathbf{do}\ print\ n\,;\ exec\ c\ (n, m)\ \} \\
& \mathbf{do}\ n \leftarrow eval\ x\,;\ exec\ (PRINT\ c)\ (n, m) \\
\lesssim_\perp & \quad \{\ \text{induction hypothesis for } x \text{ and } r\ \} \\
& exec\ (comp\ x\ r\ (PRINT\ c))\ (a, m)
\end{array}$$

This concludes the calculation of *comp* and *exec*. The top-level compiler *compile* can then be calculated from its specification (13):

$$
\begin{array}{ll}
& \mathbf{do}\ v \leftarrow \mathit{eval}\ e\,;\ \mathit{return}\ (v, \mathit{empty}) \\
= & \quad \{\ \text{define: } \mathit{exec}\ \mathit{HALT}\ s = \mathit{return}\ s\ \} \\
& \mathbf{do}\ v \leftarrow \mathit{eval}\ e\,;\ \mathit{exec}\ \mathit{HALT}\ (v, \mathit{empty}) \\
\sqsubseteq & \quad \{\ \text{specification (14) \& } \mathit{first\text{-}empty}\ \} \\
& \mathit{exec}\ (\mathit{comp}\ e\ \mathit{first}\ \mathit{HALT})\ (a, \mathit{empty})
\end{array}
$$

We can thus read off the definition $\mathit{compile}\ e = \mathit{comp}\ e\ \mathit{first}\ \mathit{HALT}$. The complete definitions of the compiler and register machine we have calculated in this section are shown in Figure 6.

To illustrate how the calculated compiler and register machine work, let's consider the expression $\mathit{Print}\ ((1 + 2) + 3)$, for which the compiler produces the following code:

$$\mathit{CONST}\ 1\ (\mathit{STORE}\ 0\ (\mathit{CONST}\ 2\ (\mathit{ADD}\ 0\ (\mathit{STORE}\ 0\ (\mathit{CONST}\ 3\ (\mathit{ADD}\ 0\ (\mathit{PRINT}\ \mathit{HALT}))))))))$$

The first *STORE* instruction stores the integer 1 in register 0. The second *STORE* instruction then overwrites register 0 with 3. The use of ordered skew bisimilarity allows the compiler to produce code that overwrites registers whose values have become unused. In particular, the reasoning step that uses the *freeFrom-set* law in the calculation for addition allows the register r to be dropped implicitly after an $\mathit{ADD}\ r$ instruction, which then can be used by the subsequent $\mathit{STORE}\ r$ instruction. This is exactly what happens with the second *STORE* instruction in the example above.

At first glance, one might think that the machine is just a stack machine in disguise: Values appear to be stored in a contiguous region in memory (similar to a stack) and we know where the next free memory location is (similar to the top of the stack). However, upon closer inspection, this is not the case: Firstly, we make no assumptions about the relative location of registers produced by *next*; they are just names. Importantly, this property makes it possible to combine the calculated compiler with a register allocation algorithm to turn register names into CPU registers and memory locations. Secondly, the next free register is only known at *compile time* via the argument to *comp*. The knowledge that a register is free is only maintained at the meta level by the *freeFrom* predicate. The machine does not maintain this information and does not "know" which registers are used or unused. This observation is formally attested by the *exec*-monotone property.

5.4 Full Compiler Correctness Property

Finally, using the fact that $\mathit{eval}\ e$ is safe for all $e :: \mathit{Expr}$, we can apply Corollary 3 to the partial correctness property we just calculated so that we can obtain the full correctness property:

$$\mathit{eval}\ e \cong \mathbf{do}\ (b, m) \leftarrow \mathit{exec}\ (\mathit{compile}\ e)\ (a, \mathit{empty})\,;\ \mathit{return}\ b \tag{15}$$

That is, evaluating e is the same as first compiling e, then running the resulting code on the machine starting with an empty memory, and finally returning the value of the accumulation register.

To derive this property, we start with the compiler specification (13) we have just proved:

$$\mathbf{do}\ v \leftarrow \mathit{eval}\ e\,;\ \mathit{return}\ (v, \mathit{empty})\ \sqsubseteq\ \mathit{exec}\ (\mathit{compile}\ e)\ (a, \mathit{empty})$$

As the left-hand side of the above inequality is a safe choice tree, we can apply Corollary 3 to obtain

$$\mathit{fmap}\ \mathit{fst}\ (\mathbf{do}\ v \leftarrow \mathit{eval}\ e\,;\ \mathit{return}\ (v, \mathit{empty})) \cong \mathit{fmap}\ \mathit{fst}\ (\mathit{exec}\ (\mathit{compile}\ e)\ (a, \mathit{empty}))$$

We can then apply the monad laws to simplify this to the full correctness property (15).

5.5 Extension to Non-termination and Concurrency

We have demonstrated the calculation technique for register machines with the help of a minimal example above. But the technique scales to more feature-rich source languages as well. The extensions discussed in section 4.4 also apply to ordered skew bisimilarity, allowing us to calculate register machine compilers for source languages featuring non-termination and concurrency. Calculations

Target language

```
data Code = CONST Int Code | STORE Reg Code
          | ADD Reg Code | PRINT Code | HALT
```

Compiler

```
compile :: Expr → Code
compile e = comp e first HALT

comp :: Expr → Reg → Code → Code
comp (Val n)     r c = CONST n c
comp (Add x y) r c = comp x r (STORE r
                        (comp y (next r) (ADD r c)))
comp (Print x)   r c = comp x r (PRINT c)
```

Register machine

```
type Conf = (Acc, Mem Int)
type Acc  = Int

exec :: Code → Conf → CTree⊥ PrintEff Conf
exec (CONST n c) (a, m) = exec c (n, m)
exec (STORE r c) (a, m) = exec c (a, set r a m)
exec (ADD r c)   (a, m) = do b ← get r m
                             exec c (b + a, m)
exec (PRINT c)   (a, m) = do print a
                             exec c (a, m)
exec HALT        s      = return s
```

Fig. 6. Compiler and register machine for the simple expression language extended with print.

of such examples can be found in the accompanying Agda formalisation [Bahr 2026]. Below we sketch the resulting definitions of one such example, namely a register machine compiler for the concurrent lambda calculus discussed in section 4.4.2, with the full definitions given in Appendix C.

Let's compare the resulting definitions from the two calculations for the concurrent lambda calculus. The main differences between the two compilers are that the register machine compiler needs to (1) keep track of the next free register, (2) insert *STORE* instructions to copy the contents of the accumulation register to another register, and (3) produce instructions with register addresses. These differences are exemplified by how the two compilers translate function application:

```
comp (App x y)   c = comp x (comp y (APP c))                    -- stack machine
comp (App x y) r c = comp x r (STORE r (comp y (next r) (APP r c)))   -- register machine
```

To compare the stack machine with the register machine, it is instructive to recall how a function $exec :: Code \to Conf \to CTree^{c}_{\bot}\ e\ Conf$ models the semantics of a (sequential) machine. The type *Conf* models the state of the machine, and since *exec* is tail recursive, executions of the machine are represented by a possibly infinite sequence $exec\ c_1\ s_1 \rightsquigarrow exec\ c_2\ s_2 \rightsquigarrow \ldots$, where each step indicated by $\rightsquigarrow$ is either an equality, i.e. applying a clause in the definition of *exec*, or a transition of the choice tree transition semantics (cf. Figure 2). That is, each term $exec\ c_i\ s_i$ represents the current state of the machine.

As observed by Bahr and Hutton [2023], for machines that use the parallel composition operator $\vec{\shortparallel}_c$ along with an effect handler *han*, the state of the machine is instead represented by

$$interpSt_c\ s\ han\ (exec\ c_1\ s_1\ \vec{\shortparallel}_c\ exec\ c_2\ s_2\ \vec{\shortparallel}_c\ \cdots\ \vec{\shortparallel}_c\ exec\ c_{n+1}\ s_{n+1})$$

That is, we have $n + 1$ parallel computations, each with their own local code c_i and state s_i. The function $interpSt_c$ applies the algebraic effect handler *han* with state s, which in this case contains the next available channel. Both the stack machine and the register machine have this same structure, but they differ in the type *Conf* of the local state s_i of each parallel computation.

```
type Conf = (Stack, Env^M)                    -- stack machine
type Conf = (Value^M, Env^M, Lam, Mem Value^M)   -- register machine
```

For the stack machine, a local state (s, γ) only consists of a stack s and an environment γ. For the register machine, a local state (a, γ, s, m) consists of an accumulation register a, an environment γ, a call stack s, and a memory m containing all local registers. This local state type for the concurrent register machine mirrors the global state of the sequential register machine that Bahr and Hutton [2020] calculated for a sequential lambda calculus.

6 Related Work

Weakened bisimilarity relations. Many different weakened forms of bisimilarity have been proposed and investigated in the concurrency theory literature [Sangiorgi 2011]. Skew bisimilarity may be considered a further weakening of the notion of *partial bisimilarity*, first introduced by Rutten [2000] in the context of controllability. Skew bisimilarity deems certain transition labels safe and weakens bisimilarity for all unsafe transition labels. In this article we considered the case where only $\uparrow \mathit{Inl}\ \mathit{Stuck}$ is considered unsafe, but this restriction is not necessary. Indeed, the Agda formalisation of skew bisimilarity [Bahr 2026] is parametrised by the set of labels that are considered unsafe. This simplifies the formalisation, since bisimilarity then becomes a special case of skew bisimilarity where the set of unsafe transition labels is empty.

Partial bisimilarity, denoted ${}_U\!\sim$, is parametrised by a set U of *uncontrollable* transition labels that play a similar role to safe transition labels for skew bisimilarity. Partial bisimilarity is defined[1] as the largest relation such that if $s\ {}_U\!\sim\ t$ then

(1) $l \in U$ and $s \overset{l}{\Longrightarrow} s'$ implies $t \overset{l}{\Longrightarrow} t'$ and $s'\ {}_U\!\sim\ t'$, and (2) $t \overset{l}{\Longrightarrow} t'$ implies $s \overset{l}{\Longrightarrow} s'$ and $s'\ {}_U\!\sim\ t'$.

However, this notion of partiality is not suitable for our purposes. We would instantiate U to be the set of safe transition labels $\{l \mid l \neq \uparrow \mathit{Inl}\ \mathit{Stuck}\}$, but we don't have the desired law $\mathit{stuck}\ {}_U\!\sim\ p$, which we do have for skew bisimilarity. Instead, we only obtain $\mathit{stuck} \oplus p\ {}_U\!\sim\ p$. That is, we may disregard unsafe behaviours on the left-hand side, but we must still account for any other behaviours.

According to Bisping et al. [2020], the name skew bisimilarity was briefly used by Robin Milner in 1995 for a different weakened form of bisimilarity. However, this notion of skew bisimilarity never found its way into the literature as it shortly afterwards turned out to be a variant of *coupled similarity* [Parrow and Sjödin 1992].

Compiler calculation. Wand [1982] pioneered the idea of deriving compilers from their specification. The central underlying techniques of Wand's approach are the use of a definitional interpreter, which is formulated using continuation-passing style, and defunctionalisation to produce the code for the target language. All of these techniques were in turn introduced by Reynolds [1972] a decade earlier and have also proved essential in later correct-by-construction compiler derivation techniques by Meijer [1992], Ager et al. [2003], and Bahr and Hutton [2015], as well as our work.

Partial correctness specification. As mentioned in section 2.1, partial specifications of compiler correctness have long been adopted by compiler verification projects [Leroy 2009]. We are not aware of a compiler *calculation* technique that employs partial correctness specifications, but Pickard and Hutton [2021] have demonstrated an elegant alternative that avoids unsafe source programs altogether. To this end, they use a dependently typed metalanguage to devise an intrinsic typing discipline that encodes the source language's type system so that the semantics, the specification, and the calculation only have to consider safe behaviours. However, so far this has only been applied to simple languages without any side effects. Devising an intrinsic typing discipline for the source language may be intractable for sophisticated type systems with subtle type safety arguments. Moreover, coming up with a corresponding type system for the target language is even more difficult as we need to anticipate how the target machine is going to work. The use of skew bisimilarity avoids these pitfalls entirely, which allows us to calculate compilers for complex languages with sophisticated type systems as we have demonstrated for the modal lambda calculus in section 4.4.1. Calculating a compiler for this language using the dependently typed approach of Pickard and Hutton [2021] would be unfeasible as the type soundness proof requires a complex

[1] To compare partial bisimilarity with skew bisimilarity, the definition given here is in fact the inverse of the partial bisimilarity relation $\sim_U$ found in the literature [Rutten 2000].

logical relations argument that is incompatible with an intrinsic typing approach. Likewise, devising a corresponding type system for the target language is a research question in its own right.

Contextual refinement. Our compiler specifications use skew bisimilarity to express that the source and target program have the *same* behaviour unless the source program is unsafe. By contrast, compiler verification efforts typically prove *behaviour refinement* similar to the one sketched in (2) in section 2.1. A program T refines S, denoted $T \sqsubseteq S$ iff any behaviour exhibited by T is also exhibited by S. Refinement is often strengthened to *contextual refinement* $\sqsubseteq_{\text{ctx}}$ by closing it under contexts, i.e. $T \sqsubseteq_{\text{ctx}} S$ iff $C[T] \sqsubseteq C[S]$ for all contexts C. Like (skew) bisimilarity, contextual refinement supports modular reasoning principles such as transitivity and congruence laws, which makes it attractive for proving compiler correctness [Neis et al. 2015; Patterson and Ahmed 2019].

However, for compiler calculation in the presence of unsafe behaviours, contextual refinement is too fine in some aspects and too coarse in other aspects. It is too fine as it considers *all* behaviours and programs, be they safe or not, and it is too coarse as it only requires the preservation of behaviours in one direction. The latter is undesirable for compiler *calculation*, since a specification that uses (contextual) refinement can be trivially satisfied by compiling all source programs to an instruction *END* with the semantics $\mathit{exec}\ \mathit{END}\ s = \mathit{Zero}$, because $\mathit{Zero} \sqsubseteq p$ for all p.

Individually, these limitations of (contextual) refinement have been addressed in the literature: Song et al. [2023] and Gäher et al. [2022] combine contextual refinement with separation logic and thus provide a rich logic to express under which circumstances a refinement holds. Moreover, some compiler verification efforts, e.g. Stewart et al. [2015], use a notion of contextual *equivalence* rather than contextual refinement in order to express that the compiler preserves all behaviours in both directions (like bisimilarity). However, we are not aware of any approach that combines these solutions to obtain an equational reasoning principle suitable for compiler calculation.

7 Conclusion and Further Work

With skew bisimilarity, we have introduced a weaker form of bisimilarity to simplify reasoning about unsafe source languages and to derive compilers that can produce more efficient code compared to calculation techniques that use a stronger notion of bisimilarity. In addition, skew bisimilarity enables a novel calculation technique to derive compilers for register machines. This constitutes a further step towards making calculation methods applicable to realistic programming languages and target machines.

Whether skew bisimilarity can be further weakened while still capturing the desired compiler correctness property remains an open question. For example, we may take *trace equivalence* as a starting point to devise a corresponding skew trace equivalence. Unlike bisimilarity, trace equivalence cannot distinguish non-deterministic behaviours that only differ in when non-deterministic choices are made. This coarser notion of equality is typically sufficient for compilers, and it does not have the drawbacks of contextual refinement discussed in section 6.

An important aspect of realistic compilers that is still missing in the compiler calculation literature is multi-stage compilers. Ongoing work on modular reasoning techniques for choice trees [Chappe et al. 2025] may provide the semantic foundation that enables a calculation technique for closing that gap. Combining the calculated register machine compilers of our work with verified register allocation algorithms would be a first step in that direction.

A further direction is the application of skew bisimilarity to compiler *verification*. The equational reasoning principles developed here are independent of whether the compiler is being *derived* from its specification or *checked* against a given implementation. Exploring this direction would provide an algebraic alternative to the simulation-based proof techniques used in large verified compiler projects, potentially making such proofs more modular and readable.

A Modal FRP language compiled to stack machine

This is a compiler for the modal FRP language from Bahr [2022].

Source language

```
data Expr = Delay Expr | Adv Expr | Out Expr
  | Num Int | Add Expr Expr | Abs Expr
  | App Expr Expr | In1 Expr | In2 Expr
  | Case Expr Expr Expr | Fix | Into Expr
  | Pair Expr Expr | Pr1 Expr | Pr2 Expr
  | Box Expr | Unbox Expr | Var Nat | Unit
```

Values

```
type Loc = Nat

data Value = Num_V Int | Clo_V Expr Env
  | Loc_V Loc | Box_V Expr Env | In1_V Value
  | In2_V Value | Pair_V Value Value
  | Into_V Value | Unit_V

type Env = [Value]
```

Algebraic effects

```
data None :: Type → Type where
```

Heaps & stores

```
type Heap a = [a]

alloc :: Heap a → a → (Loc, Heap a)
alloc η v = (length η, η ++ [v])

lookup :: Nat → [a] → CTree⊥ None a
lookup l      []      = stuck
lookup Z      (x : η) = return x
lookup (S l)  (x : η) = lookup l η

data Store a = (✓) (Heap a) (Heap a) | N (Heap a)

alloc_S :: Store a → a → (Loc, Store a)
alloc_S (η_N ✓ η_L) v = let (l, η'_L) = alloc η_L v in (l, η_N ✓ η'_L)
alloc_S (N η_N)     v = let (l, η'_N) = alloc η_N v in (l, N η'_N)

type HElem = (Expr, Env)
```

Source language semantics

```
eval :: Expr → Env → Store HElem → CTree⊥ None (Value, Store HElem)
eval (Delay t)    γ σ           = let (l, σ') = alloc_S σ (t, γ) in return (Loc_V l, σ')
eval (Adv t)      γ (η_N ✓ η_L) = do (Loc_V l, N η'_N) ← eval t γ (N η_N); (t', γ') ← lookup l η'_N
                                     Later (eval t' γ' (η'_N ✓ η_L))
eval (Adv t)      γ (N _)       = stuck
eval (Num n)      γ σ           = return (Num_V n, σ)
eval (Add t1 t2)  γ σ           = do (Num_V n1, σ1) ← eval t1 γ σ; (Num_V n2, σ2) ← eval t2 γ σ1
                                     return (Num_V (n1 + n2), σ2)
eval (Abs t)      γ σ           = return (Clo_V t γ, σ)
eval (App t1 t2)  γ σ           = do (Clo_V t1' γ', σ1) ← eval t1 γ σ; (v2, σ2) ← eval t2 γ σ1
                                     Later (eval t1' (v2 : γ') σ2)
eval Fix          γ σ           = return (Clo_V (App (Var 0) (Box (Delay (App Fix (Var 0))))) γ, σ)
eval (Var x)      γ σ           = do v ← lookup x γ; return (v, σ)
eval (Box t)      γ σ           = return (Box_V t γ, σ)
eval (Unbox t)    γ σ           = do (Box_V t' γ', σ') ← eval t γ σ; Later (eval t' γ' σ')
eval (Pair t1 t2) γ σ           = do (v1, σ1) ← eval t1 γ σ; (v2, σ2) ← eval t2 γ σ1; return (Pair_V v1 v2, σ2)
eval (Pr1 t)      γ σ           = do (Pair_V v1 v2, σ') ← eval t γ σ; return (v1, σ')
eval (Pr2 t)      γ σ           = do (Pair_V v1 v2, σ') ← eval t γ σ; return (v2, σ')
eval (In1 t)      γ σ           = do (v, σ') ← eval t γ σ; return (In1_V v, σ')
eval (In2 t)      γ σ           = do (v, σ') ← eval t γ σ; return (In2_V v, σ')
eval (Case t t1 t2) γ σ         = do (v, σ') ← eval t γ σ; case v of In1_V vl → eval t1 (vl : γ) σ'
                                                                     In2_V vr → eval t2 (vr : γ) σ'
eval (Into t)     γ σ           = do (v, σ') ← eval t γ σ; return (Into_V v, σ')
eval (Out t)      γ σ           = do (Into_V v, σ') ← eval t γ σ; return (v, σ')
eval Unit         γ σ           = return (Unit_V, σ)
```

Target language

$$\begin{aligned}&\textbf{data } \mathit{Code} = \mathit{PUSH\ Int\ Code} \mid \mathit{ADD\ Code} \mid \mathit{PAIR\ Code} \mid \mathit{PR1\ Code} \mid \mathit{PR2\ Code} \mid \mathit{IN1\ Code} \mid \mathit{IN2\ Code}\\ &\qquad \mid \mathit{ENDCASE\ Code} \mid \mathit{CASE\ Code\ Code} \mid \mathit{LOOKUP\ Nat\ Code} \mid \mathit{RET} \mid \mathit{APP\ Code}\\ &\qquad \mid \mathit{ABS\ Code\ Code} \mid \mathit{ENDADV\ Code} \mid \mathit{ADV\ Code} \mid \mathit{DELAY\ Code\ Code} \mid \mathit{UNBOX\ Code}\\ &\qquad \mid \mathit{BOX\ Code\ Code} \mid \mathit{FIX\ Code} \mid \mathit{INTO\ Code} \mid \mathit{OUT\ Code} \mid \mathit{UNIT\ Code} \mid \mathit{HALT}\end{aligned}$$

The stack machine has its own versions of values, environments, and heap elements, which we denote with the superscript M:

Stack machine

$$\begin{aligned}&\textbf{data } \mathit{Value}^M = \mathit{Num}_V^M\ \mathit{Int} \mid \mathit{Clo}_V^M\ \mathit{Code\ Env}^M \mid \mathit{Box}_V^M\ \mathit{Code\ Env}^M \mid \mathit{Loc}_V^M\ \mathit{Loc}\\ &\qquad \mid \mathit{Pair}_V^M\ \mathit{Value}^M\ \mathit{Value}^M \mid \mathit{In1}_V^M\ \mathit{Value}^M \mid \mathit{In2}_V^M\ \mathit{Value}^M \mid \mathit{Into}_V^M\ \mathit{Value}^M \mid \mathit{Unit}_V^M\\ &\textbf{type } \mathit{Env}^M = [\,\mathit{Value}^M\,]\\ &\textbf{data } \mathit{Elem} = \mathit{VAL\ Value}^M \mid \mathit{HEAP}\ (\mathit{Heap}\ (\mathit{Code}, \mathit{Env}^M)) \mid \mathit{CLO\ Code\ Env}^M\\ &\textbf{type } \mathit{Stack} = [\,\mathit{Elem}\,]\\ &\textbf{type } \mathit{HElem}^M = (\mathit{Code}, \mathit{Env}^M)\\ &\textbf{type } \mathit{Conf} = (\mathit{Stack}, \mathit{Env}^M, \mathit{Store\ HElem}^M)\\ &\mathit{exec} :: \mathit{Code} \to \mathit{Conf} \to \mathit{CTree}_\perp\ \mathit{None\ Conf}\\ &\mathit{exec}\ (\mathit{PUSH}\ n\ c)\ (s, \gamma, \sigma) = \mathit{exec}\ c\ (\mathit{VAL}\ (\mathit{Num}_V^M\ n) : s, \gamma, \sigma)\\ &\mathit{exec}\ (\mathit{ADD}\ c)\ (\mathit{VAL}\ (\mathit{Num}_V^M\ n) : \mathit{VAL}\ (\mathit{Num}_V^M\ m) : s, \gamma, \sigma) = \mathit{exec}\ c\ (\mathit{VAL}\ (\mathit{Num}_V^M\ (m+n)) : s, \gamma, \sigma)\\ &\mathit{exec}\ (\mathit{PAIR}\ c)\ (\mathit{VAL}\ v_2 : \mathit{VAL}\ v_1 : s, \gamma, \sigma) = \mathit{exec}\ c\ (\mathit{VAL}\ (\mathit{Pair}_V^M\ v_1\ v_2) : s, \gamma, \sigma)\\ &\mathit{exec}\ (\mathit{PR1}\ c)\ (\mathit{VAL}\ (\mathit{Pair}_V^M\ v_1\ v_2) : s, \gamma, \sigma) = \mathit{exec}\ c\ (\mathit{VAL}\ v_1 : s, \gamma, \sigma)\\ &\mathit{exec}\ (\mathit{PR2}\ c)\ (\mathit{VAL}\ (\mathit{Pair}_V^M\ v_1\ v_2) : s, \gamma, \sigma) = \mathit{exec}\ c\ (\mathit{VAL}\ v_2 : s, \gamma, \sigma)\\ &\mathit{exec}\ (\mathit{IN1}\ c)\ (\mathit{VAL}\ v : s, \gamma, \sigma) = \mathit{exec}\ c\ (\mathit{VAL}\ (\mathit{In1}_V^M\ v) : s, \gamma, \sigma)\\ &\mathit{exec}\ (\mathit{IN2}\ c)\ (\mathit{VAL}\ v : s, \gamma, \sigma) = \mathit{exec}\ c\ (\mathit{VAL}\ (\mathit{In2}_V^M\ v) : s, \gamma, \sigma)\\ &\mathit{exec}\ (\mathit{LOOKUP}\ n\ c)\ (s, \gamma, \sigma) = \textbf{do}\ v \leftarrow \mathit{lookup}\ n\ \gamma\\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{exec}\ c\ (\mathit{VAL}\ v : s, \gamma, \sigma)\\ &\mathit{exec}\ \mathit{RET}\ (\mathit{VAL}\ u : \mathit{CLO}\ c\ \gamma' : s, \_, \sigma) = \mathit{exec}\ c\ (\mathit{VAL}\ u : s, \gamma', \sigma)\\ &\mathit{exec}\ (\mathit{APP}\ c)\ (\mathit{VAL}\ v : \mathit{VAL}\ (\mathit{Clo}_V^M\ c'\ \gamma') : s, \gamma, \sigma) = \mathit{Later}\ (\mathit{exec}\ c'\ (\mathit{CLO}\ c\ \gamma : s, v : \gamma', \sigma))\\ &\mathit{exec}\ (\mathit{ABS}\ c'\ c)\ (s, \gamma, \sigma) = \mathit{exec}\ c\ (\mathit{VAL}\ (\mathit{Clo}_V^M\ c'\ \gamma) : s, \gamma, \sigma)\\ &\mathit{exec}\ \mathit{HALT}\ c = \mathit{return}\ c\\ &\mathit{exec}\ (\mathit{ADV}\ c)\ (s, \gamma, \eta_N \checkmark \eta_L) = \mathit{exec}\ c\ (\mathit{HEAP}\ \eta_L : s, \gamma, N\ \eta_N)\\ &\mathit{exec}\ (\mathit{ENDADV}\ c)\ (\mathit{VAL}\ (\mathit{Loc}_V^M\ l) : \mathit{HEAP}\ \eta_L : s, \gamma, N\ \eta_N) = \textbf{do}\ (c', \gamma') \leftarrow \mathit{lookup}\ l\ \eta_N\\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{Later}\ (\mathit{exec}\ c'\ (\mathit{CLO}\ c\ \gamma : s, \gamma', \eta_N \checkmark \eta_L))\\ &\mathit{exec}\ (\mathit{DELAY}\ c'\ c)\ (s, \gamma, \sigma) = \textbf{let}\ (l, \sigma') = \mathit{alloc}_S\ \sigma\ (c', \gamma)\\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ \mathit{exec}\ c\ (\mathit{VAL}\ (\mathit{Loc}_V^M\ l) : s, \gamma, \sigma')\\ &\mathit{exec}\ (\mathit{UNBOX}\ c)\ (\mathit{VAL}\ (\mathit{Box}_V^M\ c'\ \gamma') : s, \gamma, \sigma) = \mathit{Later}\ (\mathit{exec}\ c'\ (\mathit{CLO}\ c\ \gamma : s, \gamma', \sigma))\\ &\mathit{exec}\ (\mathit{BOX}\ c'\ c)\ (s, \gamma, \sigma) = \mathit{exec}\ c\ (\mathit{VAL}\ (\mathit{Box}_V^M\ c'\ \gamma) : s, \gamma, \sigma)\\ &\mathit{exec}\ (\mathit{FIX}\ c)\ (s, \gamma, \sigma) = \mathit{exec}\ c\ (\mathit{VAL}\ v : s, \gamma, \sigma)\\ &\quad \textbf{where}\ v = \mathit{Clo}_V^M\ (\mathit{LOOKUP}\ 0\ (\mathit{BOX}\ (\mathit{DELAY}\ (\mathit{FIX}\ (\mathit{LOOKUP}\ 0\ (\mathit{APP\ RET})))\ \mathit{RET})\ (\mathit{APP\ RET})))\ \gamma\\ &\mathit{exec}\ (\mathit{ENDCASE}\ c)\ (s, v : \gamma, \sigma) = \mathit{exec}\ c\ (s, \gamma, \sigma)\\ &\mathit{exec}\ (\mathit{CASE}\ c_1\ c_2)\ (\mathit{VAL}\ (\mathit{In1}_V^M\ v) : s, \gamma, \sigma) = \mathit{exec}\ c_1\ (s, v : \gamma, \sigma)\\ &\mathit{exec}\ (\mathit{CASE}\ c_1\ c_2)\ (\mathit{VAL}\ (\mathit{In2}_V^M\ v) : s, \gamma, \sigma) = \mathit{exec}\ c_2\ (s, v : \gamma, \sigma)\\ &\mathit{exec}\ (\mathit{INTO}\ c)\ (\mathit{VAL}\ v : s, \gamma, \sigma) = \mathit{exec}\ c\ (\mathit{VAL}\ (\mathit{Into}_V^M\ v) : s, \gamma, \sigma)\\ &\mathit{exec}\ (\mathit{OUT}\ c)\ (\mathit{VAL}\ (\mathit{Into}_V^M\ v) : s, \gamma, \sigma) = \mathit{exec}\ c\ (\mathit{VAL}\ v : s, \gamma, \sigma)\\ &\mathit{exec}\ (\mathit{UNIT}\ c)\ (s, \gamma, \sigma) = \mathit{exec}\ c\ (\mathit{VAL}\ \mathit{Unit}_V^M : s, \gamma, \sigma)\\ &\mathit{exec}\ \_\ \_ = \mathit{stuck}\end{aligned}$$

Compiler

```
comp :: Expr → Code → Code
comp (Num n)    c = PUSH n c
comp (Add x y)  c = comp x (comp y (ADD c))
comp (Adv x)    c = ADV (comp x (ENDADV c))
comp (Delay x)  c = DELAY (comp x RET) c
comp (Unbox x)  c = comp x (UNBOX c)
comp (Box x)    c = BOX (comp x RET) c
comp (App x y)  c = comp x (comp y (APP c))
comp (Abs x)    c = ABS (comp x RET) c
comp (Var n)    c = LOOKUP n c
comp Unit       c = UNIT c
comp Fix        c = FIX c
comp (Pair x y)     c = comp x (comp y (PAIR c))
comp (Pr1 x)        c = comp x (PR1 c)
comp (Pr2 x)        c = comp x (PR2 c)
comp (In1 x)        c = comp x (IN1 c)
comp (In2 x)        c = comp x (IN2 c)
comp (Into x)       c = comp x (INTO c)
comp (Out x)        c = comp x (OUT c)
comp (Case x x1 x2) c
  = comp x (CASE (comp x1 (ENDCASE c))
                 (comp x2 (ENDCASE c)))

compile :: Expr → Code
compile e = comp e HALT
```

B Concurrent lambda calculus compiled to stack machine

This is a compiler for the concurrent lambda calculus from Bahr and Hutton [2023]. The syntax and semantics of the source language are the same as in Bahr and Hutton [2023].

Source language

```
data Expr = App Expr Expr | Abs Expr
          | Val Int | Add Expr Expr
          | Send Expr Expr | Receive Expr
          | Fork Expr | Var Nat
```

Values

```
lookup :: Nat → [a] → CTree^c_⊥ e a
lookup _     []      = stuck_c
lookup Z     (x : γ) = return x
lookup (S n) (x : γ) = lookup n γ

data Value = Num Int | Clo Expr Env

type Env = [Value]
```

Algebraic effects

```
type Chan = Int

data ChanEff where
  SendInt    :: Chan → Int → ChanEff ()
  ReceiveInt :: Chan → ChanEff Int
  NewChan    :: ChanEff Chan

send :: Chan → Int → CTree^c_⊥ ChanEff ()
send tid n = Eff_c (Inr (SendInt tid n)) return

receive :: Chan → CTree^c_⊥ ChanEff Int
receive tid = Eff_c (Inr (ReceiveInt tid)) return

newChan :: CTree^c_⊥ ChanEff Chan
newChan = Eff_c (Inr NewChan) return
```

Semantics

```
eval :: Expr → Env → CTree^c_⊥ ChanEff Value
eval (Val x)      γ = return (Num x)
eval (Add x y)    γ = do Num n ← eval x γ
                         Num m ← eval y γ
                         return (Num (n + m))
eval (Fork x)     γ = do ch ← newChan
                         (eval x (Num ch : γ)
                           ⫛_c return (Num ch))
eval (Send x y)   γ = do Num ch ← eval x γ
                         Num n ← eval y γ
                         send ch n
                         return (Num n)
eval (Receive x)  γ = do Num ch ← eval x γ
                         n ← receive ch
                         return (Num n)
eval (Abs x)      γ = return (Clo x γ)
eval (Var n)      γ = lookup n γ
eval (App x y)    γ = do Clo x′ γ′ ← eval x γ
                         v ← eval y γ
                         Later_c (eval x′ (v : γ′))

hanChan :: Chan → ChanEff b
                → CTree^c_⊥ None (b, Chan)
hanChan ch (SendInt _ _) = Zero_c
hanChan ch (ReceiveInt _) = Zero_c
hanChan ch NewChan       = return (ch, ch + 1)

evaluate :: Expr → CTree^c_⊥ None Value
evaluate x = interpSt_c 0 hanChan (eval x [])
```

Target language

$$\textbf{data}\ \mathit{Code} = \mathit{PUSH}\ \mathit{Int}\ \mathit{Code} \mid \mathit{ADD}\ \mathit{Code} \mid \mathit{LOOKUP}\ \mathit{Nat}\ \mathit{Code} \mid \mathit{RET} \mid \mathit{APP}\ \mathit{Code} \mid \mathit{ABS}\ \mathit{Code}\ \mathit{Code} \mid \mathit{SEND}\ \mathit{Code} \mid \mathit{RECEIVE}\ \mathit{Code} \mid \mathit{FORK}\ \mathit{Code}\ \mathit{Code} \mid \mathit{HALT}$$

Stack machine

$$\textbf{data}\ \mathit{Value}^M = \mathit{Num}_V^M\ \mathit{Int} \mid \mathit{Clo}_V^M\ \mathit{Code}\ \mathit{Env}^M$$

$$\textbf{type}\ \mathit{Env}^M = [\,\mathit{Value}^M\,]$$

$$\textbf{data}\ \mathit{Elem} = \mathit{VAL}\ \mathit{Value}^M \mid \mathit{CLO}\ \mathit{Code}\ \mathit{Env}^M$$

$$\textbf{type}\ \mathit{Stack} = [\,\mathit{Elem}\,]$$

$$\textbf{type}\ \mathit{Conf} = (\mathit{Stack}, \mathit{Env}^M)$$

$$\begin{aligned}
&\mathit{exec} :: \mathit{Code} \to \mathit{Conf} \to \mathit{CTree}^{\mathrm{c}}_{\perp}\ \mathit{ChanEff}\ \mathit{Conf}\\
&\mathit{exec}\ (\mathit{PUSH}\ n\ c)\ \ (s,\gamma) &&= \mathit{exec}\ c\ (\mathit{VAL}\ (\mathit{Num}_V^M\ n) : s, \gamma)\\
&\mathit{exec}\ (\mathit{ADD}\ c)\ \ (\mathit{VAL}\ (\mathit{Num}_V^M\ n) : \mathit{VAL}\ (\mathit{Num}_V^M\ m) : s, \gamma) &&= \mathit{exec}\ c\ (\mathit{VAL}\ (\mathit{Num}_V^M\ (m+n)) : s, \gamma)\\
&\mathit{exec}\ (\mathit{LOOKUP}\ n\ c)\ (s,\gamma) &&= \textbf{do}\ v \leftarrow \mathit{lookup}\ n\ \gamma\\
&&&\qquad \mathit{exec}\ c\ (\mathit{VAL}\ v : s, \gamma)\\
&\mathit{exec}\ (\mathit{ABS}\ c'\ c)\ \ (s,\gamma) &&= \mathit{exec}\ c\ (\mathit{VAL}\ (\mathit{Clo}_V^M\ c'\ \gamma) : s, \gamma)\\
&\mathit{exec}\ \mathit{RET}\ \ (\mathit{VAL}\ u : \mathit{CLO}\ c\ \gamma' : s, _) &&= \mathit{exec}\ c\ (\mathit{VAL}\ u : s, \gamma')\\
&\mathit{exec}\ (\mathit{APP}\ c)\ \ (\mathit{VAL}\ v : \mathit{VAL}\ (\mathit{Clo}_V^M\ c'\ \gamma') : s, \gamma) &&= \mathit{Later}_{\mathrm{c}}\ (\mathit{exec}\ c'\ (\mathit{CLO}\ c\ \gamma : s, v : \gamma'))\\
&\mathit{exec}\ (\mathit{SEND}\ c)\ \ (\mathit{VAL}\ (\mathit{Num}_V^M\ n) : \mathit{VAL}\ (\mathit{Num}_V^M\ ch) : s, \gamma) &&= \textbf{do}\ \mathit{send}\ ch\ n\\
&&&\qquad \mathit{exec}\ c\ (\mathit{VAL}\ (\mathit{Num}_V^M\ n) : s, \gamma)\\
&\mathit{exec}\ (\mathit{RECEIVE}\ c)\ \ (\mathit{VAL}\ (\mathit{Num}_V^M\ ch) : s, \gamma) &&= \textbf{do}\ n \leftarrow \mathit{receive}\ ch\\
&&&\qquad \mathit{exec}\ c\ (\mathit{VAL}\ (\mathit{Num}_V^M\ n) : s, \gamma)\\
&\mathit{exec}\ (\mathit{FORK}\ c'\ c)\ \ (s,\gamma) &&= \textbf{do}\ ch \leftarrow \mathit{newChan}\\
&&&\qquad (\mathit{exec}\ c'\ ([\,], \mathit{Num}_V^M\ ch : \gamma)\\
&&&\qquad\quad \vec{\amalg}_{\mathrm{c}}\ \mathit{exec}\ c\ (\mathit{VAL}\ (\mathit{Num}_V^M\ ch) : s, \gamma))\\
&\mathit{exec}\ \mathit{HALT}\ \ (s,\gamma) &&= \mathit{return}\ (s,\gamma)\\
&\mathit{exec}\ _\ \ _ &&= \mathit{stuck}_{\mathrm{c}}
\end{aligned}$$

$$\begin{aligned}
&\mathit{execute} :: \mathit{Code} \to \mathit{Stack} \to \mathit{CTree}^{\mathrm{c}}_{\perp}\ \mathit{None}\ \mathit{Conf}\\
&\mathit{execute}\ c\ s = \mathit{interpSt}_{\mathrm{c}}\ 0\ \mathit{hanChan}\ (\mathit{exec}\ c\ (s, [\,]))
\end{aligned}$$

Compiler

$$\begin{aligned}
&\mathit{comp} :: \mathit{Expr} \to \mathit{Code} \to \mathit{Code}\\
&\mathit{comp}\ (\mathit{Val}\ n)\ \ c = \mathit{PUSH}\ n\ c &&\mathit{comp}\ (\mathit{Abs}\ x)\ \ c = \mathit{ABS}\ (\mathit{comp}\ x\ \mathit{RET})\ c\\
&\mathit{comp}\ (\mathit{Add}\ x\ y)\ c = \mathit{comp}\ x\ (\mathit{comp}\ y\ (\mathit{ADD}\ c)) &&\mathit{comp}\ (\mathit{Send}\ x\ y)\ c = \mathit{comp}\ x\ (\mathit{comp}\ y\ (\mathit{SEND}\ c))\\
&\mathit{comp}\ (\mathit{Var}\ i)\ \ c = \mathit{LOOKUP}\ i\ c &&\mathit{comp}\ (\mathit{Receive}\ x)\ c = \mathit{comp}\ x\ (\mathit{RECEIVE}\ c)\\
&\mathit{comp}\ (\mathit{App}\ x\ y)\ c = \mathit{comp}\ x\ (\mathit{comp}\ y\ (\mathit{APP}\ c)) &&\mathit{comp}\ (\mathit{Fork}\ x)\ \ c = \mathit{FORK}\ (\mathit{comp}\ x\ \mathit{HALT})\ c
\end{aligned}$$

$$\begin{aligned}
&\mathit{compile} :: \mathit{Expr} \to \mathit{Code}\\
&\mathit{compile}\ e = \mathit{comp}\ e\ \mathit{HALT}
\end{aligned}$$

C Concurrent lambda calculus compiled to register machine

The syntax and semantics of the source language are the same as in Appendix B, but the compiler targets a register machine.

Target language

$$\textbf{data}\ \mathit{Code} = \mathit{CONST}\ \mathit{Int}\ \mathit{Code} \mid \mathit{STORE}\ \mathit{Reg}\ \mathit{Code} \mid \mathit{ADD}\ \mathit{Reg}\ \mathit{Code} \mid \mathit{LOOKUP}\ \mathit{Nat}\ \mathit{Code} \mid \mathit{APP}\ \mathit{Reg}\ \mathit{Code} \mid \mathit{RET} \mid \mathit{ABS}\ \mathit{Code}\ \mathit{Code} \mid \mathit{SEND}\ \mathit{Reg}\ \mathit{Code} \mid \mathit{RECEIVE}\ \mathit{Code} \mid \mathit{FORK}\ \mathit{Code}\ \mathit{Code} \mid \mathit{HALT}$$

Register machine

$$\textbf{data}\ \mathit{Value}^M = \mathit{Num}_V^M\ \mathit{Int} \mid \mathit{Clo}_V^M\ \mathit{Code}\ \mathit{Env}^M \qquad \textbf{type}\ \mathit{Env}^M = [\,\mathit{Value}^M\,]$$
$$\textbf{type}\ \mathit{Conf} = (\mathit{Value}^M, \mathit{Env}^M, \mathit{Lam}, \mathit{Mem}\ \mathit{Value}^M) \qquad \textbf{type}\ \mathit{Lam} = [\,\mathit{Mem}\ \mathit{Value}^M\,]$$

$$\begin{aligned}
&\mathit{exec} :: \mathit{Code} \to \mathit{Conf} \to \mathit{CTree}^{\mathrm{c}}_{\perp}\ \mathit{ChanEff}\ \mathit{Conf}\\
&\mathit{exec}\ (\mathit{CONST}\ n\ c)\ \ (a,\gamma,l,m) &&= \mathit{exec}\ c\ (\mathit{Num}_V^M\ n,\gamma,l,m)\\
&\mathit{exec}\ (\mathit{ADD}\ r\ c)\ \ (\mathit{Num}_V^M\ a,\gamma,l,m) &&= \textbf{do}\ (\mathit{Num}_V^M\ b) \leftarrow \mathit{get}\ r\ m\\
&&&\qquad \mathit{exec}\ c\ (\mathit{Num}_V^M\ (b+a),\gamma,l,m)\\
&\mathit{exec}\ (\mathit{STORE}\ r\ c)\ \ (a,\gamma,l,m) &&= \mathit{exec}\ c\ (a,\gamma,l,\mathit{set}\ r\ a\ m)\\
&\mathit{exec}\ (\mathit{LOOKUP}\ n\ c)\ (a,\gamma,l,m) &&= \textbf{do}\ v \leftarrow \mathit{lookup}\ n\ \gamma\\
&&&\qquad \mathit{exec}\ c\ (v,\gamma,l,m)\\
&\mathit{exec}\ (\mathit{APP}\ r\ c)\ \ (a,\gamma,l,m) &&= \textbf{do}\ \mathit{Clo}_V^M\ c'\ \gamma' \leftarrow \mathit{get}\ r\ m\\
&&&\qquad \mathit{Later}_{\mathrm{c}}\ (\mathit{exec}\ c'\ (a, a:\gamma', m:l, \mathit{set\ first}\ (\mathit{Clo}_V^M\ c\ \gamma)\ \mathit{empty}))\\
&\mathit{exec}\ (\mathit{ABS}\ c'\ c)\ \ (a,\gamma,l,m) &&= \mathit{exec}\ c\ (\mathit{Clo}_V^M\ c'\ \gamma,\gamma,l,m)\\
&\mathit{exec}\ \mathit{RET}\ \ (a,\gamma,m':l,m) &&= \textbf{do}\ \mathit{Clo}_V^M\ c'\ \gamma' \leftarrow \mathit{get\ first}\ m\\
&&&\qquad \mathit{exec}\ c'\ (a,\gamma',l,m')\\
&\mathit{exec}\ (\mathit{SEND}\ r\ c)\ \ (\mathit{Num}_V^M\ n,\gamma,l,m) &&= \textbf{do}\ \mathit{Num}_V^M\ \mathit{ch} \leftarrow \mathit{get}\ r\ m\\
&&&\qquad \mathit{send}\ \mathit{ch}\ n\\
&&&\qquad \mathit{exec}\ c\ (\mathit{Num}_V^M\ n,\gamma,l,m)\\
&\mathit{exec}\ (\mathit{RECEIVE}\ c)\ \ (\mathit{Num}_V^M\ \mathit{ch},\gamma,l,m) &&= \textbf{do}\ n \leftarrow \mathit{receive}\ \mathit{ch}\\
&&&\qquad \mathit{exec}\ c\ (\mathit{Num}_V^M\ n,\gamma,l,m)\\
&\mathit{exec}\ (\mathit{FORK}\ c'\ c)\ \ (a,\gamma,l,m) &&= \textbf{do}\ \mathit{ch} \leftarrow \mathit{newChan}\\
&&&\qquad (\mathit{exec}\ c'\ (\mathit{Num}_V^M\ 0, \mathit{Num}_V^M\ \mathit{ch}:\gamma, [\,], \mathit{empty})\\
&&&\qquad\quad \vec{\amalg}_{\mathrm{c}}\ \mathit{exec}\ c\ (\mathit{Num}_V^M\ \mathit{ch},\gamma,l,m))\\
&\mathit{exec}\ \mathit{HALT}\ \ s &&= \mathit{return}\ s\\
&\mathit{exec}\ _\ \ _ &&= \mathit{stuck}_{\mathrm{c}}
\end{aligned}$$

$$\begin{aligned}
&\mathit{execute} :: \mathit{Code} \to \mathit{Value}^M \to \mathit{CTree}^{\mathrm{c}}_{\perp}\ \mathit{None}\ \mathit{Conf}\\
&\mathit{execute}\ c\ a = \mathit{interpSt}_{\mathrm{c}}\ 0\ \mathit{hanChan}\ (\mathit{exec}\ c\ (a, [\,], [\,], \mathit{empty}))
\end{aligned}$$

Compiler

$$\begin{aligned}
&\mathit{comp} :: \mathit{Expr} \to \mathit{Reg} \to \mathit{Code} \to \mathit{Code}\\
&\mathit{comp}\ (\mathit{Val}\ n)\ \ r\ c = \mathit{CONST}\ n\ c\\
&\mathit{comp}\ (\mathit{Add}\ x\ y)\ \ r\ c = \mathit{comp}\ x\ r\ (\mathit{STORE}\ r\ (\mathit{comp}\ y\ (\mathit{next}\ r)\ (\mathit{ADD}\ r\ c)))\\
&\mathit{comp}\ (\mathit{Var}\ n)\ \ r\ c = \mathit{LOOKUP}\ n\ c\\
&\mathit{comp}\ (\mathit{App}\ x\ y)\ \ r\ c = \mathit{comp}\ x\ r\ (\mathit{STORE}\ r\ (\mathit{comp}\ y\ (\mathit{next}\ r)\ (\mathit{APP}\ r\ c)))\\
&\mathit{comp}\ (\mathit{Abs}\ x)\ \ r\ c = \mathit{ABS}\ (\mathit{comp}\ x\ (\mathit{next\ first})\ \mathit{RET})\ c\\
&\mathit{comp}\ (\mathit{Send}\ x\ y)\ \ r\ c = \mathit{comp}\ x\ r\ (\mathit{STORE}\ r\ (\mathit{comp}\ y\ (\mathit{next}\ r)\ (\mathit{SEND}\ r\ c)))\\
&\mathit{comp}\ (\mathit{Receive}\ x)\ r\ c = \mathit{comp}\ x\ r\ (\mathit{RECEIVE}\ c)\\
&\mathit{comp}\ (\mathit{Fork}\ x)\ \ r\ c = \mathit{FORK}\ (\mathit{comp}\ x\ \mathit{first}\ \mathit{HALT})\ c
\end{aligned}$$

$$\begin{aligned}
&\mathit{compile} :: \mathit{Expr} \to \mathit{Code}\\
&\mathit{compile}\ x = \mathit{comp}\ x\ \mathit{first}\ \mathit{HALT}
\end{aligned}$$

References

Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. 2003. *From Interpreter to Compiler and Virtual Machine: A Functional Derivation.* Technical Report RS-03-14. BRICS, Department of Computer Science, University of Aarhus.

Roland Backhouse. 2003. *Program Construction: Calculating Implementations from Specifications.* John Wiley and Sons, Inc.

Patrick Bahr. 2022. Modal FRP for all: Functional reactive programming without space leaks in Haskell. *Journal of Functional Programming* 32 (2022), e15. doi:10.1017/S0956796822000132

Patrick Bahr. 2026. Supplementary Material for "Safety First: How to Safely Disregard Unsafe Behaviour in Compiler Calculations". doi:10.5281/zenodo.20415074

Patrick Bahr and Graham Hutton. 2015. Calculating correct compilers. *Journal of Functional Programming* 25 (2015). doi:10.1017/S0956796815000180

Patrick Bahr and Graham Hutton. 2020. Calculating correct compilers II: Return of the register machines. *Journal of Functional Programming* 30 (2020). doi:10.1017/S0956796820000209

Patrick Bahr and Graham Hutton. 2022. Monadic compiler calculation (functional pearl). *Proceedings of the ACM on Programming Languages* 6, ICFP (2022). doi:10.1145/3547624

Patrick Bahr and Graham Hutton. 2023. Calculating Compilers for Concurrency. *Proceedings of the ACM on Programming Languages* 7, ICFP (2023), 213:740–213:767. doi:10.1145/3607855

R. S. Bird. 1989a. Algebraic Identities for Program Calculation. *The Computer Journal* 32, 2 (1989), 122–126. doi:10.1093/comjnl/32.2.122

Richard S. Bird. 1989b. Lectures on Constructive Functional Programming. In *Constructive Methods in Computing Science*. 151–217. doi:10.1007/978-3-642-74884-4_5

Benjamin Bisping, Uwe Nestmann, and Kirstin Peters. 2020. Coupled similarity: the first 32 years. *Acta Informatica* 57, 3 (2020), 439–463. doi:10.1007/s00236-019-00356-4

Edwin Brady. 2017. *Type-Driven Development with Idris.* Manning Publications.

Nicolas Chappe, Paul He, Ludovic Henrio, Eleftherios Ioannidis, Yannick Zakowski, and Steve Zdancewic. 2025. Choice trees: Representing and reasoning about nondeterministic, recursive, and impure programs in Rocq. *Journal of Functional Programming* 35 (2025), e21. doi:10.1017/S0956796825100105

Ranald Clouston. 2018. Fitch-Style Modal Lambda Calculi. In *Foundations of Software Science and Computation Structures*. 258–275. doi:10.1007/978-3-319-89366-2_14

Nils Anders Danielsson and Thorsten Altenkirch. 2010. Subtyping, Declaratively. In *Proceedings of the International Conference on Mathematics of Program Construction*. doi:10.1007/978-3-642-13321-3_8

Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. 2022. Simuliris: a separation logic framework for verifying concurrent program optimizations. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 28:1–28:31. doi:10.1145/3498689

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. doi:10.1145/2578855.2535841

Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Communications of the ACM* 52, 7 (2009). doi:10.1145/1538788.1538814

Erik Meijer. 1992. *Calculating Compilers.* Ph. D. Dissertation. Katholieke Universiteit Nijmegen.

Carroll Morgan. 1994. *Programming from specifications (2nd ed.).* Prentice Hall International (UK) Ltd.

Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. 2015. Pilsner: A Compositionally Verified Compiler for a Higher-order Imperative Language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. 166–178. doi:10.1145/2784731.2784764

Ulf Norell. 2007. *Towards a Practical Programming Language Based on Dependent Type Theory.* PhD Thesis. Chalmers University of Technology.

Joachim Parrow and Peter Sjödin. 1992. Multiway synchronization verified with coupled simulation. In *CONCUR '92*. 518–533. doi:10.1007/BFb0084813

Daniel Patterson and Amal Ahmed. 2019. The Next 700 Compiler Correctness Theorems (Functional Pearl). *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 85:1–85:29. doi:10.1145/3341689

Mitchell Pickard and Graham Hutton. 2021. Calculating dependently-typed compilers (functional pearl). *Proceedings of the ACM on Programming Languages* 5, ICFP (2021), 82:1–82:27. doi:10.1145/3473587

John C Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM Annual Conference*. doi:10.1145/800194.805852

J. J. M. M. Rutten. 2000. Coalgebra, Concurrency, and Control. In *Discrete Event Systems: Analysis and Control*. 31–38. doi:10.1007/978-1-4615-4493-7_2

Davide Sangiorgi. 2011. *Introduction to bisimulation and coinduction.* Cambridge University Press. doi:10.1017/CBO9780511777110

Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer. 2023. Conditional Contextual Refinement. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 39:1121–39:1151. doi:10.1145/3571232

Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 275–287. doi:10.1145/2676726.2676985

Wouter Swierstra. 2008. Data types à la carte. *Journal of Functional Programming* 18, 4 (2008), 423–436. doi:10.1017/S0956796808006758

Janis Voigtländer. 2008. Asymptotic Improvement of Computations over Free Monads. In *Proceedings of the International Conference on Mathematics of Program Construction*. doi:10.1007/978-3-540-70594-9_20

Mitchell Wand. 1982. Deriving Target Code as a Representation of Continuation Semantics. *ACM Transactions on Programming Languages and Systems* 4, 3 (1982). doi:10.1145/357172.357179

Received 2026-02-19; accepted 2026-05-13