Partial Compiler Calculation with Skew Bisimilarity

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 of 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 been long 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 more efficient compilers while at the same time obtaining the same strong correctness guarantees for safe source programs.

We further show that – even for 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 limited to terminating source languages without effects. To demonstrate the effectiveness of skew bisimilarity as a proof technique, we 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.

1 INTRODUCTION

Compilers have long been recognised as an ideal target for formal verification [McCarthy and Painter 1967] as they are complex pieces of software and constitute a critical link in any secure software system. *Program calculation* is a technique to derive correct-by-construction programs from specifications of their desired behaviour [Backhouse 2003]. Applied to compilers, this technique allows us to *discover* new compilation techniques along with proofs of their correctness.

In its simplest form, the specification of a compiler is an equation of the form

$$[\![e]\!] \cong [\![compile\ e]\!]$$

which states that the semantics $[\![e]\!]$ of any source program e must be the same as the semantics of the compiled program $compile\ e$. The bisimilarity relation $\cong$ demands that both sides behave in the same way.

However, in practice we don't expect compilers to satisfy the above equation for *all* source programs, but only those programs that are safe. Discounting unsafe behaviours of the source program in the specification affords 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 ${}_{\perp}{\cong}$, that allows us to formulate a *partial* compiler specification in the form of an *inequality*

$$[\![e]\!]\ {}_{\perp}{\cong}\ [\![compile\ e]\!]$$

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, we show that any compiler that satisfies a partial specification of this form then also satisfies the full correctness

Author's address: Patrick Bahr, IT University of Copenhagen, Denmark.

property for all safe source programs:

$$[\![e]\!] \cong [\![\mathit{compile}\ e]\!] \qquad \text{whenever } e \text{ is safe}$$

The use of skew bisimilarity not only simplifies the equational reasoning for compiler calculations involving unsafe languages, it also results in compilers that exploit the less strict specification to generate more efficient code.

We also show that skew bisimilarity has further applications beyond source languages with unsafe behaviours. Even in cases where the source language is safe, the use of a partial specification enables a new compiler calculation technique that allows us 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 still obtain the full correctness property in terms of bisimilarity.

We demonstrate this new compiler calculation technique enabled by skew bisimilarity on variety of example languages. For the purpose of exposition, we focus on simple toy languages in this article. In addition, the supplementary material contains Agda formalisations of compiler calculations for more realistic languages, including lambda calculi with different features such as algebraic effects, concurrency, and modal type operators for reactive programming.

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 examples we need the richer semantic structure provided by choice trees [Chappe et al. 2023], which we introduce in section 3. In section 4, we then introduce the skew bisimilarity relation on choice trees and apply it to a simple expression 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 meta language for all programs and calculations, but we assume that the meta language is total. All definitions, theorems, and compiler calculations presented in this article have been formalised in Agda [Norell 2007] and are available as supplementary material.

2 PARTIAL SEMANTICS

In this section, we show how to calculate a compiler in the presence of unsafe or undefined 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 an expression to a value:

```
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 = x in if b then eval y else eval z
```

The addition operator expects its two arguments to evaluate to a number, whereas the conditional operator expects a Boolean as its first argument. The definition uses non-exhaustive pattern matching and thus evaluation fails on ill-typed expression. For example, $\mathit{eval}\ (\mathit{Add}\ (\mathit{Val}\ (B\ \mathit{True}))\ (\mathit{Val}\ (N\ 9)))$ is not defined. The above definition of *eval* only works if the meta language were not total. Instead of relying on a non-total meta language, 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:

$$\begin{array}{l}
\mathbf{data}\ \mathit{Maybe}\ a = \mathit{Just}\ a \mid \mathit{Nothing} \\[4pt]
\mathit{return} :: a \rightarrow \mathit{Maybe}\ a \\
\mathit{return} = \mathit{Just} \\[4pt]
(\gg\!=) :: \mathit{Maybe}\ a \rightarrow (a \rightarrow \mathit{Maybe}\ b) \rightarrow \mathit{Maybe}\ b \\
\mathit{Nothing} \gg\!= f = \mathit{Nothing} \\
\mathit{Just}\ x \quad \gg\!= f = f\ x
\end{array}$$

With the *Maybe* monad in hand we can revise the interpreter *eval* so that a type mismatch produces the result value *Nothing*:

$$\begin{array}{l}
\mathit{eval} :: \mathit{Expr} \rightarrow \mathit{Maybe}\ \mathit{Value} \\
\mathit{eval}\ (\mathit{Val}\ v) \qquad = \mathit{return}\ v \\
\mathit{eval}\ (\mathit{Add}\ x\ y) = \mathbf{do}\ N\ m \leftarrow \mathit{eval}\ x;\ N\ n \leftarrow \mathit{eval}\ y;\ \mathit{return}\ (N\ (m+n)) \\
\mathit{eval}\ (\mathit{If}\ \ c\ x\ y)\ = \mathbf{do}\ B\ b \ \ \leftarrow c;\ \mathbf{if}\ b\ \mathbf{then}\ \mathit{eval}\ x\ \mathbf{else}\ \mathit{eval}\ y
\end{array}$$

To reduce syntactic clutter, we use Haskell's **do** notation, which translates into applications of $\gg\!=$. In addition, we use the fact that *Maybe* is an instance of the *MonadFail* type class, which in case of a non-exhaustive pattern matching failing invokes its *fail* method

$$\begin{array}{l}
\mathit{fail} :: \mathit{String} \rightarrow \mathit{Maybe}\ a \\
\mathit{fail}\ \_ = \mathit{Nothing}
\end{array}$$

For example, the clause for $\mathit{eval}\ (\mathit{If}\ \ c\ x\ y)$ desugars into

$$\begin{array}{l}
\mathit{eval}\ (\mathit{If}\ \ c\ x\ y) = c \gg\!= \lambda v \rightarrow \mathbf{case}\ v\ \mathbf{of}\ B\ b \rightarrow \mathbf{if}\ b\ \mathbf{then}\ \mathit{eval}\ x\ \mathbf{else}\ \mathit{eval}\ y \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \_ \quad\ \rightarrow \mathit{Nothing}
\end{array}$$

With the revised *eval* function, $\mathit{Add}\ \ (\mathit{Val}\ \ (B\ \ \mathit{True}))\ \ (\mathit{Val}\ \ (N\ \ 9))$ now evaluates to *Nothing*: The first argument of *Add* evaluates to $B\ \mathit{True}$ and thus the partial pattern match with $N\ m$ fails, resulting in the value *Nothing*, which in turn is propagated by the definition of the monadic bind operator $\gg\!=$.

2.1 Compiler Correctness

Our goal is to calculate a compiler $\mathit{compile} :: \mathit{Expr} \rightarrow \mathit{Code}$ that translates an expression into an as-of-yet unspecified target language. To this end, we assume that the compiler targets a stack machine, whose semantics is given by a function $\mathit{exec} :: \mathit{Code} \rightarrow \mathit{Stack} \rightarrow \mathit{Maybe}\ \mathit{Stack}$, where $\mathit{Stack} = [\,\mathit{Value}]$. In order to account for unsafe behaviour, also the semantics of the target language is given in terms of the *Maybe* monad.

All three 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 executed after the compiled code. This will significantly simplify the calculation process [Bahr and Hutton 2015]. Using this idea, the specification for the generalised compilation function $\mathit{comp} :: \mathit{Expr} \rightarrow \mathit{Code} \rightarrow \mathit{Code}$ is as follows:

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

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 or undefined behaviour to perform optimisations. Leroy [2009] expresses this intuition of the relation between a source program S and compiled program C as follows

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

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, $\text{Wrong} = \{\mathit{Nothing}\}$.

Assuming that the target language has a deterministic semantics, Leroy then strengthens the above specification into a form that is easier to prove:

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

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

$$\textbf{do}\ v \leftarrow \mathit{eval}\ e\,;\ \mathit{exec}\ c\ (v : s) = \mathit{Just}\ w \quad\implies\quad \mathit{exec}\ (\mathit{comp}\ e\ c)\ s = \mathit{Just}\ w \tag{2}$$

To perform a calculation we need to rephrase this specification into an equation or at least into an inequality so that we can perform equational reasoning. We achieve this by defining an ordering ${}_{\perp}\!\!=$ on *Maybe* types:

$$\mathit{Nothing}\ {}_{\perp}\!\!=\ p \qquad \text{for any } p :: \mathit{Maybe}\ a \qquad ({}_{\perp}\!\!= \text{-}\mathit{Nothing})$$

$$\mathit{Just}\ v\ {}_{\perp}\!\!=\ \mathit{Just}\ v \qquad \text{for any } v :: a. \qquad ({}_{\perp}\!\!= \text{-}\mathit{Just})$$

Using this ordering on *Maybe*, the above specification (2) can now be reformulated equivalently as

$$\textbf{do}\ v \leftarrow \mathit{eval}\ e\,;\ \mathit{exec}\ c\ (v : s)\ \ {}_{\perp}\!\!=\ \ \mathit{exec}\ (\mathit{comp}\ e\ c)\ s \tag{3}$$

Since ${}_{\perp}\!\!=$ 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 [2020]. The crucial property of ${}_{\perp}\!\!=$ 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. ${}_{\perp}\!\!=$ according to the defining equation (${}_{\perp}\!\!=$ -*Nothing*). In addition, we will make extensive use of the monad laws:

$$\mathit{return}\ x \gg\!= f = f\ x \qquad \text{(left identity)}$$

$$\mathit{mx} \gg\!= \mathit{return} = \mathit{mx} \qquad \text{(right identity)}$$

$$(\mathit{mx} \gg\!= f) \gg\!= g = \mathit{mx} \gg\!= (\lambda x\,.\,(f\ x \gg\!= g)) \qquad \text{(associativity)}$$

2.2 Compiler Calculation

We can now calculate the desired compiler for the source language. To this end, we prove specification (3) 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

$$\textbf{do}\ v \leftarrow \mathit{eval}\ e\,;\ \mathit{exec}\ c\ (v : s)\ \ {}_{\perp}\!\!=\ \ \mathit{exec}\ c'\ s$$

We can then set $comp\ e\ c = c'$ 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 virtual machine *exec* along with new constructors for *Code*.

We start with the case for $e = Val\ n$, which is straightforward:

$$
\begin{array}{ll}
 & \mathbf{do}\ v \leftarrow eval\ (Val\ n)\,;\ exec\ c\ (v : s) \\
= & \{\ \text{definition of } eval\ \} \\
 & \mathbf{do}\ v \leftarrow return\ n\,;\ exec\ c\ (v : s) \\
= & \{\ \text{monad laws (left identity)}\ \} \\
 & exec\ c\ (n : s) \\
= & \{\ \text{define: } exec\ (PUSH\ n\ c)\ s = exec\ c\ (n : s)\ \} \\
 & exec\ (PUSH\ n\ c)\ s
\end{array}
$$

In the first two steps, we simply apply the definition of *eval* and simplify the term using the left identity law of the *Maybe* monad. We then aim to turn the term $exec\ c\ (n : s)$ into the desired form $exec\ c'\ s$. That is, we must solve the equation $exec\ c'\ s = exec\ c\ (n : s)$. We can do so by turning this equation into a defining clause for *exec*. To this end, we must ensure that all variables on the right-hand side also occur on the left-hand side. Variable s already occurs on the left-hand side. The remaining variables n and c can be packaged up in the argument c' with the help of a new constructor $PUSH :: Int \rightarrow Code \rightarrow Code$ that takes them as argument. We then solve the equation by setting $c' = PUSH\ n\ c$ and thus adding the following clause to the definition of *exec*:

$$exec\ (PUSH\ n\ c)\ s = exec\ c\ (n : s)$$

That is, we have discovered the first instruction of the virtual machine and its semantics. Moreover, with the above calculation we arrive at the desired form, namely $exec\ c'\ s$, and can thus read off the first clause of the compiler:

$$comp\ (Val\ n)\ c = PUSH\ n\ c$$

We proceed with the case for $e = Add\ x\ y$, which starts in a similar manner: We first apply the definition of *eval* followed by the monad laws. Our goal is to manipulate the term so that we can apply the induction hypothesis. This requires solving an equation, which we do by again introducing a defining clause for *exec*:

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

However, now we appear to be stuck, since the subterm

$$\mathbf{do}\ N\ n \leftarrow eval\ y\,;\ exec\ (ADD\ c)\ (N\ n : N\ m : s)$$

does not quite match the induction hypothesis for y. To see why, let's unfold the syntactic sugar for the non-exhaustive pattern matching:

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

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 easily achieved by appealing to the ${}_{\perp}\!=$ *-Nothing* law to conclude that $\mathit{Nothing} \;{}_{\perp}\!=\; \mathit{exec}\ (\mathit{ADD}\ c)\ (v : N\ m : s)$:

$$
\begin{array}{ll}
& \mathbf{do}\ v \leftarrow \mathit{eval}\ y;\ \mathbf{case}\ v\ \mathbf{of}\ N\ n \rightarrow \mathit{exec}\ (\mathit{ADD}\ c)\ (N\ n : N\ m : s) \\
& \phantom{\mathbf{do}\ v \leftarrow \mathit{eval}\ y;\ \mathbf{case}\ v\ \mathbf{of}\ N}\ \_ \ \ \rightarrow \mathit{Nothing} \\
{}_{\perp}\!= & \{\ {}_{\perp}\!= \textit{-Nothing}\ \text{law}\ \} \\
& \mathbf{do}\ v \leftarrow \mathit{eval}\ y;\ \mathbf{case}\ v\ \mathbf{of}\ N\ n \rightarrow \mathit{exec}\ (\mathit{ADD}\ c)\ (N\ n : N\ m : s) \\
& \phantom{\mathbf{do}\ v \leftarrow \mathit{eval}\ y;\ \mathbf{case}\ v\ \mathbf{of}\ N}\ \_ \ \ \rightarrow \mathit{exec}\ (\mathit{ADD}\ c)\ (v : N\ m : s) \\
= & \{\ \text{case analysis on}\ v\ \} \\
& \mathbf{do}\ v \leftarrow \mathit{eval}\ y;\ \mathit{exec}\ (\mathit{ADD}\ c)\ (v : N\ m : s)
\end{array}
$$

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

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

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

$$
\begin{array}{ll}
& \mathbf{do}\ v \leftarrow \mathit{eval}\ (\mathit{If}\ x\ y\ z);\ \mathit{exec}\ c\ (v : s) \\
= & \{\ \text{definition of}\ \mathit{eval}\ \} \\
& \mathbf{do}\ v \leftarrow \mathbf{do}\ \{B\ b \leftarrow \mathit{eval}\ x;\ \mathbf{if}\ b\ \mathbf{then}\ \mathit{eval}\ y\ \mathbf{else}\ \mathit{eval}\ z\};\ \mathit{exec}\ c\ (v : s) \\
= & \{\ \text{monad laws (left identity \& associativity)}\ \} \\
& \mathbf{do}\ B\ b \leftarrow \mathit{eval}\ b;\ \mathbf{if}\ b\ \mathbf{then}\ (\mathbf{do}\ v \leftarrow \mathit{eval}\ y;\ \mathit{exec}\ c\ (v : s)) \\
& \phantom{\mathbf{do}\ B\ b \leftarrow \mathit{eval}\ b;\ \mathbf{if}\ b\ }\mathbf{else}\ \ (\mathbf{do}\ v \leftarrow \mathit{eaval}\ z;\ \mathit{exec}\ c\ (v : s)) \\
{}_{\perp}\!= & \{\ \text{induction hypothesis for}\ y\ \text{and}\ z\ \} \\
& \mathbf{do}\ B\ b \leftarrow \mathit{eval}\ x;\ \mathbf{if}\ b\ \mathbf{then}\ \mathit{exec}\ (\mathit{comp}\ x\ c)\ s \\
& \phantom{\mathbf{do}\ B\ b \leftarrow \mathit{eval}\ x;\ \mathbf{if}\ b\ }\mathbf{else}\ \ \mathit{exec}\ (\mathit{comp}\ y\ c)\ s \\
= & \{\ \text{define:}\ \mathit{exec}\ (\mathit{JPC}\ c'\ c)\ (B\ b : s) = \mathbf{if}\ b\ \mathbf{then}\ \mathit{exec}\ c'\ s\ \mathbf{else}\ \mathit{exec}\ c\ s\ \} \\
& \mathbf{do}\ B\ b \leftarrow \mathit{eval}\ x;\ \mathit{exec}\ (\mathit{JPC}\ (\mathit{comp}\ x\ c)\ (\mathit{comp}\ y\ c))\ (B\ b : s) \\
{}_{\perp}\!= & \{\ {}_{\perp}\!= \textit{-Nothing}\ \text{law}\ \} \\
& \mathbf{do}\ v \leftarrow \mathit{eval}\ x;\ \mathit{exec}\ (\mathit{JPC}\ (\mathit{comp}\ x\ c)\ (\mathit{comp}\ y\ c))\ (v : s) \\
{}_{\perp}\!= & \{\ \text{induction hypothesis for}\ x\ \} \\
& \mathit{exec}\ (\mathit{comp}\ b\ (\mathit{JPC}\ (\mathit{comp}\ x\ c)\ (\mathit{comp}\ y\ c)))\ s
\end{array}
$$

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

Target language

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

Compiler

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

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

Virtual 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.

$$comp\ (If\ x\ y\ z)\ c = comp\ b\ (JPC\ (comp\ x\ c)\ (comp\ y\ c))$$

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

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

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

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

In summary, we have calculated the definitions in Figure 1.

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

For this very simple language, the use of the weaker specification was not necessary as we would also be able to calculate the same compiler and virtual 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 partial compiler specification will allow us to calculate a more efficient compiler.

3 CHOICE TREES

In section 2, we used the *Maybe* monad to express undefined behaviour of our source language. This approach is applicable when we have a language without effects, i.e. the only observable behaviour is the final result of the evaluation. To generalise this calculation approach to languages with effects such as non-termination, non-determinism and algebraic effects, we replace the *Maybe* monad with *choice trees* [Bahr and Hutton 2023]. 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* represents non-deterministic computations that return values of type *a* and that may use algebraic effects described by the type function $e :: * \to *$:

```
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 returns the value *v* without performing any effects, $p \oplus q$ 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 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*). We adopt here the ∞ notation for mixed inductive-coinductive types in Agda [Danielsson and Altenkirch 2010], but to reduce clutter we assume conversions *Delay* :: *CTree e a* → ∞ (*CTree e a*) and *Force* :: ∞ (*CTree e a*) → *CTree* 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*, *Step* and so on, 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:

$$\begin{aligned}&\mathit{return} :: a \to \mathit{CTree}\ e\ a\\&\mathit{return} = \mathit{Now}\end{aligned}$$

$$\begin{aligned}&(\gg\!=) :: \mathit{CTree}\ e\ a \to (a \to \mathit{CTree}\ e\ b) \to \mathit{CTree}\ e\ b\\&\mathit{Now}\ v\ \ \gg\!= f = f\ v\\&(p \oplus q) \gg\!= f = (p \gg\!= f) \oplus (q \gg\!= f)\\&\mathit{Zero}\quad \gg\!= f = \mathit{Zero}\\&\mathit{Eff}\ o\ c\ \gg\!= f = \mathit{Eff}\ o\ (\lambda i \to c\ i \gg\!= f)\\&\mathit{Later}\ p \gg\!= f = \mathit{Later}\ (p \gg\!= f)\end{aligned}$$

Similarly to the *Maybe* monad, we will use Haskell's do notation to make choice tree definitions more readable.

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

data *Stuck a* **where**
  Stuck :: *Stuck Void*

where *Void* is the empty data type. We use the name *Stuck* since we can think of this effect as representing a computation that is stuck. Using *Stuck*, we can form the type *CTree Stuck a* of computations that may get stuck:

$$\begin{aligned}&\mathit{stuck} :: \mathit{CTree}\ \mathit{Stuck}\ a\\&\mathit{stuck} = \mathit{Eff}\ \mathit{Stuck}\ \mathit{elimVoid}\end{aligned}$$

where $\mathit{elimVoid} :: \mathit{Void} \to a$ is the eliminator for the empty type.

Like the *Maybe* monad, we can make *CTree Stuck* an instance of the *MonadFail* type class

$$\begin{aligned}&\mathit{fail} :: \mathit{String} \to \mathit{CTree}\ \mathit{Stuck}\ a\\&\mathit{fail}\ \_ = \mathit{stuck}\end{aligned}$$

We can now rewrite the evaluator from section 2 to use *CTree Stuck* instead of *Maybe*. In fact, definition of *eval* itself remains syntactically exactly the same and only its type signature changes:

$$\begin{aligned}&\mathit{eval} :: \mathit{Expr} \to \mathit{CTree}\ \mathit{Stuck}\ \mathit{Value}\\&\mathit{eval}\ (\mathit{Val}\ v)\qquad = \mathit{return}\ v\\&\mathit{eval}\ (\mathit{Add}\ x\ y) = \mathbf{do}\ N\ m \leftarrow \mathit{eval}\ x;\ N\ n \leftarrow \mathit{eval}\ y;\ \mathit{return}\ (N\ (m+n))\\&\mathit{eval}\ (\mathit{If}\ c\ x\ y) = \mathbf{do}\ \ B\ b\ \ \leftarrow c;\ \mathbf{if}\ b\ \mathbf{then}\ \mathit{eval}\ x\ \mathbf{else}\ \mathit{eval}\ y\end{aligned}$$

While the definition of *eval* remains unchanged, we now have a much richer semantic framework that allows us to express effectful computations. Let's extend the expression language with a simple effect by including an operator that prints the value of an integer expression:

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

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:

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

$$\frac{}{\mathit{Now}\ v \overset{\langle v\rangle}{\Longrightarrow} \mathit{Zero}} \qquad \frac{p \overset{l}{\Longrightarrow} p'}{p \oplus q \overset{l}{\Longrightarrow} p'} \qquad \frac{q \overset{l}{\Longrightarrow} q'}{p \oplus q \overset{l}{\Longrightarrow} q'}$$

$$\frac{o :: e\ b \quad c :: b \to \mathit{CTree}\ e\ a}{\mathit{Eff}\ o\ c \overset{\uparrow o}{\Longrightarrow} c} \qquad \frac{c :: b \to \mathit{CTree}\ e\ a \quad i :: b}{c \overset{\downarrow i}{\Longrightarrow} c\ i} \qquad \frac{}{\mathit{Later}\ p \overset{\tau}{\Longrightarrow} p}$$

Fig. 2. Transition semantics for choice trees.

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. *Stuck* $\prec$ *Stuck* $\uplus$ *PrintEff*. In particular, $\prec$ provides an injection function:

$\mathit{inj} :: e \prec f \Rightarrow e\ a \to f\ a$

This allows us to generalise the *print* and *stuck* effect, as well as the *fail* method of the *MonadFail* instance:

$\mathit{stuck} :: \mathit{Stuck} \prec e \Rightarrow \mathit{CTree}\ e\ a$
$\mathit{stuck} = \mathit{Eff}\ (\mathit{inj}\ \mathit{Stuck})\ \mathit{elimVoid}$

$\mathit{print} :: \mathit{PrintEff} \prec e \Rightarrow \mathit{Int} \to \mathit{CTree}\ e\ ()$
$\mathit{print}\ n = \mathit{Eff}\ (\mathit{inj}\ (\mathit{PrintInt}\ n))\ \mathit{Now}$

$\mathit{fail} :: \mathit{Stuck} \prec e \Rightarrow \mathit{String} \to \mathit{CTree}\ e\ a$
$\mathit{fail}\ \_ = \mathit{stuck}$

Finally, we can define the semantics of the extended expression language with print effect:

$\mathit{eval} :: \mathit{Expr} \to \mathit{CTree}\ (\mathit{Stuck} \uplus \mathit{PrintEff})\ \mathit{Value}$
$\mathit{eval}\ (\mathit{Val}\ v) \quad = \mathit{return}\ v$
$\mathit{eval}\ (\mathit{Add}\ x\ y) = \mathbf{do}\ N\ m \leftarrow \mathit{eval}\ x;\ N\ n \leftarrow \mathit{eval}\ y;\ \mathit{return}\ (N\ (m+n))$
$\mathit{eval}\ (\mathit{If}\ c\ x\ y) = \mathbf{do}\ B\ b \leftarrow c;\ \mathbf{if}\ b\ \mathbf{then}\ \mathit{eval}\ x\ \mathbf{else}\ \mathit{eval}\ y$
$\mathit{eval}\ (\mathit{Print}\ x) \ = \mathbf{do}\ N\ n \leftarrow \mathit{eval}\ x;\ \mathit{print}\ n;\ \mathit{return}\ (N\ n)$

To calculate a compiler for this extended expression language we need to generalise the ordering $_\bot\!\!=$ 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

The semantics of choice trees is given by the labelled transition system proposed by Chappe et al. [2023] and revised by Bahr and Hutton [2023]. A state in this labelled transition system consists either of 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 value v of type 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. The definition of the labelled transition system is shown in Figure 2.

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

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

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

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

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:

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

Since effect transitions $\uparrow o$ and input transitions $\downarrow i$ always occur in immediate succession, we typically write them as a single transition labelled $\uparrow o \downarrow i$. For example, we write the above transition sequence more compactly as

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

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 $\mathit{elimVoid} \xRightarrow{l} p$ or $\mathit{Zero} \xRightarrow{l} p$.

3.3 Bisimilarity

Because the semantics of choice trees *CTree e a* 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:

If $p \cong q$ and $p \xRightarrow{l} p'$, then there is some q' with $q \xRightarrow{l} q'$ and $p' \cong q'$

If $p \cong q$ and $q \xRightarrow{l} q'$, then there is some p' with $p \xRightarrow{l} p'$ and $p' \cong q'$

We have that the type of choice trees forms a monad modulo bisimilarity:

$$\begin{aligned} \mathit{return}\ x \gg= f &\cong f\ x \\ p \gg= \mathit{return} &\cong p \\ (p \gg= f) \gg= g &\cong p \gg= \lambda x \to (f\ x \gg= g) \end{aligned}$$

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:

$$\frac{p \cong q \qquad f\ x \cong g\ x \ \text{ for all } x}{p \gg= f \ \cong\ q \gg= g}$$

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 the following unit law for *Zero* and $\oplus$:

$$\mathit{Zero} \oplus p \cong p$$

When defining operational semantics 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 not reachable. Even though *Zero* is a ‘stuck’ choice tree in the sense that it has no outgoing transitions, the above law 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:

$$\mathbf{do}\ v \leftarrow \mathit{eval}\ e\,;\ \mathit{exec}\ c\ (v : s) \ \cong\ \mathit{exec}\ (\mathit{comp}\ e\ c)\ s \tag{4}$$

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

We proceed by induction on e, and try to manipulate the left-hand side of the specification (4) into the form $\mathit{exec}\ c'\ s$. The case for $e = \mathit{Val}\ v$ is straightforward as it is essentially the same as the calculation in section 2. The case for $e = \mathit{Add}\ x\ y$ starts as follows:

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

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

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

We are now in the same situation where we seek to manipulate the term so that we can apply the induction hypothesis. Again we can achieve this by introducing a new equation for *exec*. But now this requires the insertion of a new instruction:

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

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

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

This additional instruction is required to obtain the strong correctness property (4) in general. We abort calculation here and instead seek to revise the specification.

In the next section, we introduce a weaker bisimilarity relation $\perp\!\!\cong$ that corresponds to the $\perp\!\!=$ relation on *Maybe* and which satisfies a law corresponding to the $\perp\!\!=$ -*Nothing* law that we were sorely missing in the above calculation. While this weakens the compiler correctness property, we still obtain the strong correctness property 4 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 following shorthand $CTree_\perp$, which we refer to as the type of *partial choice trees*:

$\textbf{type}\ CTree_\perp\ e\ a = CTree\ (Stuck \uplus e)\ a$

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

1. If $p\ {}_\perp\!\cong\ q$, p is locally safe, and $p \xRightarrow{l} p'$, then there is some $q \xRightarrow{l} q'$ with $p'\ {}_\perp\!\cong\ q'$.
2. If $p\ {}_\perp\!\cong\ q$, p is locally safe, and $q \xRightarrow{l} q'$, then there is some $p \xRightarrow{l} p'$ with $p'\ {}_\perp\!\cong\ q'$.

That is, $p\ {}_\perp\!\cong\ q$ vacuously holds if p is not locally safe. In particular, this means that skew bisimilarity satisfies the following additional law:

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

This corresponds to the ${}_\perp\!=$ *-Nothing* law for the order ${}_\perp\!=$ on the *Maybe* monad and like the ${}_\perp\!=$ *-Nothing* law, the ${}_\perp\!\cong$ *-stuck* law allows us to discard unsafe behaviour in compiler calculations.

Finally, 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:

1. If $p :: CTree_\perp\ e\ a$ is safe, then p is locally safe and any q with $p \xRightarrow{l} q$ is safe.
2. If $c :: b \to CTree_\perp\ e\ a$ is safe, then $c\ i$ is safe for all $i :: b$.

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

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

1. *If $p \cong q$, then $p\ {}_\perp\!\cong\ q$.*
2. *If p is safe and $p\ {}_\perp\!\cong\ q$, 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, also all congruence laws hold for skew bisimilarity. 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 conditionals from section 3. Similarly to the aborted calculation in section 3, we aim to calculate a compiler $comp :: Expr \to Code \to Code$ for a stack machine $exec :: Code \to Stack \to CTree_\perp\ PrintEff\ Stack$. However, how we weaken the compiler correctness specification by using skew bisimilarity instead of full bisimilarity:

$$\textbf{do}\ v \leftarrow eval\ e;\ exec\ c\ (v : s)\ \ {}_\perp\!\cong\ \ exec\ c'\ s \qquad (5)$$

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 ${}_\perp\!=$ *-Nothing* law for the *Maybe* monad, the calculation uses the monad laws and the ${}_\perp\!\cong$ *-stuck* law of choice trees, respectively. We only have to consider the remaining case for $e = \mathit{Print}\ x$, which follows in a similar manner:

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

Hence, we can read off the definition of compiler for this case as $\mathit{comp}\ c\ (\mathit{Print}\ x) = \mathit{comp}\ x\ (\mathit{PRINT}\ c)$.

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

$$\mathbf{do}\ v \leftarrow \mathit{eval}\ e\,;\ \mathit{return}\ (v : s) \quad {}_{\perp}\!\cong \quad \mathit{exec}\ (\mathit{compile}\ e)\ s \tag{6}$$

The calculation proceeds in the same way as in section 2. The resulting complete definitions of the compiler and virtual machine calculated in this fashion are shown in Figure 3.

The use of the partial correctness property (3) 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 calculate a compiler that produces 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.

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

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

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 more efficient compiler calculated in this section transforms the above expression into

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

for which we observe the following transitions:

$$
\begin{array}{rl}
& \mathit{exec}\ (\mathit{PUSH}\ (B\ \mathit{True})\ (\mathit{PUSH}\ (N\ 1)\ (\mathit{PRINT}\ (\mathit{ADD}\ \mathit{HALT}))))\ [\,] \\
& = \mathit{exec}\ (\mathit{PUSH}\ (N\ 1)\ (\mathit{PRINT}\ (\mathit{ADD}\ \mathit{HALT})))\ [B\ \mathit{True}] \\
& = \mathit{exec}\ (\mathit{PRINT}\ (\mathit{ADD}\ \mathit{HALT}))\ [N\ 1, B\ \mathit{True}] \\
\xRightarrow{\uparrow \mathit{Inr}\ (\mathit{PrintInt}\ 1)\ \downarrow()} & \mathit{exec}\ (\mathit{ADD}\ \mathit{HALT})\ [N\ 1, B\ \mathit{True}] \xRightarrow{\uparrow \mathit{Inl}\ \mathit{Stuck}} \ldots
\end{array}
$$

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 4 does not hold: the semantics of the source expression is *not* bisimilar to the semantics of the compiled expression.

Target language

data *Code* = *PUSH Int Code* | *ADD Code* | *JPC Code Code* | *PRINT Code* | *HALT*

Compiler

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

comp :: Expr → Code → Code
comp (Val n)      c = PUSH n c
comp (Add x y)  c = comp x (comp y (ADD c))
comp (If x y z)  c = comp b (JPC (comp x c) (comp y c))
comp (Print x)   c = comp x (PRINT c)
```

Virtual machine

```
type Stack = [Value]
exec :: Code → Stack → CTree⊥ PrintEff 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 (PRINT c)  (N n : s)          = do print n; exec c (N n : s)
exec HALT        s                  = return s
exec _           _                  = stuck
```

Fig. 3. Compiler and stack machine for the simple expression language extended with print.

By contrast the compiler from section 3.4 compiles the above expression into the code

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

for which we observe the following transitions:

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

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.

4.2 Type safety

The compiler specification (5) only captures partial correctness. However, we can strengthen this partial correctness property to the strict specification (4) if we restrict ourselves to well-typed

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

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

Fig. 4. Proof rules for P-safety.

expressions. To this end, we consider a simple type system:

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

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

Our goal is to invoke Proposition 1 to strengthen the compiler correctness property (5) to

$$\textbf{do}\ v \leftarrow \mathit{eval}\ e;\ \mathit{return}\ (v : s) \quad \cong \quad \mathit{exec}\ (\mathit{compile}\ e)\ s \qquad \text{if } \vdash e : \tau \tag{7}$$

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

$$\mathit{eval}\ e \ \text{ is safe for all } \ \vdash e : \tau \tag{8}$$

To prove this by induction on e, we need to strengthen the property so that it additionally states that *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* is 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 (8) to the following:

$$\begin{gathered} \mathit{eval}\ e \ \text{ is } P_\tau\text{-safe for all } \ \vdash e : \tau, \\ \text{where } P_\tau(v) \text{ iff } \vdash_{\mathsf{v}} v : \tau \end{gathered} \tag{9}$$

To prove this we may use the proof rules shown in Figure 4. The rule for *Later* is denoted with a double line to indicate that it is a coinductive rule. The proof of (9) proceeds by a straightforward induction on the derivation of $\vdash e : \tau$. For example, in the case for $\vdash \mathit{If}\ c\ x\ y : \tau$, we have by induction hypothesis that $\mathit{eval}\ c$ is P_{Bool}-safe. Hence, by the proof rule for $\gg\!=$, it remains to be shown that **if** b **then** $\mathit{eval}\ y$ **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, we have the compiler correctness property (7).

4.3 Extensions

We demonstrated the compiler calculation technique based on skew bisimilarity for a very simple source language. However, this technique applies also to more realistic languages. We will only sketch some of these examples and refer the reader to the supplementary material that contains Agda formalisations of all examples mentioned in this section.

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* since $\cong_i$ enjoys the following congruence rule for *Later*

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

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 formulate skew bisimilarity also in a step-indexed version, denoted ${}_\perp{\cong}_i$. While ${}_\perp{\cong}$ is defined coinductively, ${}_\perp{\cong}_i$ is defined inductively as the smallest relation such that $p\ {}_\perp{\cong}_0\ q$ holds, and moreover $p\ {}_\perp{\cong}_{i+1}\ q$ holds if the following two conditions hold (where $j = i$ if $l = \tau$, and $j = i+1$ otherwise):

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

This step-indexed version of skew bisimilarity satisfies the same congruence laws as step-indexed bisimilarity. In particular, we have the following congruence law for *Later* that allows us to reason about non-terminating languages:

$$\frac{p\ {}_\perp{\cong}_j\ q \quad \text{for all } j < i}{\mathit{Later}\ p\ \ {}_\perp{\cong}_i\ \ \mathit{Later}\ q}$$

Concurrency. Choice trees can be used to give semantics to programming languages with concurrency features [Chappe et al. 2023]. 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 wraps choice trees inside the *codensity monad* [Voigtländer 2008]:

type $\mathit{CTree}_c\ e\ a = \mathit{forall}\ r\,.\,(a \to \mathit{CTree}\ e\ r) \to \mathit{CTree}\ e\ r$

As Bahr and Hutton [2023] show, the operations on choice trees – $\gg\!=$, *return*, $\oplus$, etc. – can be lifted to codensity choice trees. Likewise, the (step-indexed) bisimilarity relation can be lifted to codensity choice trees by simply quantifying over all continuations:

$$p \cong_i q \quad \text{iff} \quad p\ c \cong_i q\ c \ \text{ for all } c$$

The notion of (step-indexed) skew bisimilarity can be lifted in the same fashion:

$$p\ {}_\perp{\cong}_i\ q \quad \text{iff} \quad p\ c\ {}_\perp{\cong}_i\ q\ c \ \text{ for all } c$$

We tested step-indexed skew bisimilarity, by calculating a compiler for the same concurrent lambda calculus as Bahr and Hutton [2023] considered. By using step-indexed skew bisimilarity, we can derive a more efficient compiler. Similar to the compiler we sketched in section 3.4, the compiler calculated by Bahr and Hutton has to insert instructions similar to *ISN* whose only purpose is to type check values at runtime to satisfy the overly strict correctness property. Our new calculation avoids this inefficiency while still obtaining the full correctness property for well-typed terms.

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 instead of stack machines.

We expect an efficient compiler to produce code that simply overwrites unused registers. 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 ordering $\sqsubseteq$ 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 may have freed a register that is read from later. That is why we need to be able to reason about unsafe behaviours even in the context of 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.

5.1 Reasoning about Register Machines

We begin by recalling the abstract memory model of Bahr and Hutton [2020] rephrased slightly to match our semantic framework. The memory model consists of an abstract type *Reg* of registers and a type *Mem a* of memory instances with registers containing values of type *a* along with a set of operations on them:

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

Intuitively, *first* is the first register, and *next* produces a fresh register given an existing register. In turn, *empty* produces an empty memory, *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 further contains the partial order $\sqsubseteq$ on *Mem a* and the 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 the memory m.

$$\mathit{freeFrom}\ \mathit{first}\ \mathit{empty} \qquad (\mathit{first}\text{-}\mathit{empty})$$

$$\mathit{get}\ r\ (\mathit{set}\ r\ v\ m) = \mathit{return}\ v \qquad (\mathit{set}\text{-}\mathit{get})$$

$$\mathit{freeFrom}\ r\ m \Rightarrow m \sqsubseteq \mathit{set}\ r\ v\ m \qquad (\mathit{freeFrom}\text{-}\mathit{set})$$

$$\mathit{freeFrom}\ r\ m \Rightarrow \mathit{freeFrom}\ (\mathit{next}\ r)\ (\mathit{set}\ r\ v\ m) \qquad (\mathit{freeFrom}\text{-}\mathit{next})$$

$$m \sqsubseteq m' \Rightarrow \mathit{set}\ r\ v\ m \sqsubseteq \mathit{set}\ r\ v\ m' \qquad (\mathit{set}\text{-monotone})$$

$$m \sqsubseteq m' \Rightarrow \mathit{get}\ r\ m \ {}_{\bot}{\cong}\ \mathit{get}\ r\ m' \qquad (\mathit{get}\text{-monotone})$$

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

5.2 Ordered Skew Bisimilarity

Let's reconsider the expression language from section 3 whose semantics is given by $eval :: Expr \to CTree_\bot\ PrintEff\ Value$. Using the above memory model, our aim is to derive a compiler $compile :: Expr \to Code$ along with a virtual machine of type $exec :: Code \to Conf \to CTree_\bot\ PrintEff\ Conf$ that instead of a stack, operates on machine configurations of type $Conf$ defined as follows:

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

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 we extend the partial order $\sqsubseteq$ to machine configurations:

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

Bahr and Hutton [2020] only considered pure computations, i.e. the semantics is given by a function of type $eval :: Expr \to Value$ and the machine of type $exec :: Code \to Conf \to Conf$. In this simplified setting, the compiler correctness property can be phrased as follows:

$$(eval\ e, empty) \sqsubseteq exec\ (compile\ e)\ (a, empty) \tag{10}$$

The use of the $\sqsubseteq$ ordering means that we allow the right-hand side machine configuration 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.

As Bahr and Hutton [2020] have demonstrated, this 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 also have 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.

To combine this compiler calculation technique with choice trees, we need to generalise skew bisimilarity relation ${}_\bot\!\cong$ so that it also incorporates the order $\sqsubseteq$ on machine configurations. To define such a relation on partial choice trees of type $CTree_\bot\ e\ a$, we assume an order $\sqsubseteq$ on a and extend it to the set of labels in the labelled transition system. The ordering $\sqsubseteq$ 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 ${}_\bot\!\lesssim$, as the largest relation that satisfies the following:

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

Compared 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, $return\ v\ {}_\bot\!\lesssim\ return\ v'$ holds whenever $v \sqsubseteq v'$. With the help of the ordered skew bisimilarity relation ${}_\bot\!\lesssim$ we can now formulate the compiler correctness property for register machines within the semantic framework of choice trees.

When reasoning with ordered skew bisimilarity, we have the same congruence laws at our disposal as for skew bisimilarity. In addition, we have the following relation between skew bisimilarity and ordered skew bisimilarity:

Proposition 2. *Let $p, q :: CTree_\bot\ e\ a$ be two partial choice trees.*

(1) *If $p\ {}_\bot\!\cong\ q$, then $p\ {}_\bot\!\lesssim\ q$.*
(2) *If $x \sqsubseteq y$ implies $x = y$ for all $x, y :: a$, then $p\ {}_\bot\!\lesssim\ q$ implies $p\ {}_\bot\!\cong\ q$.*

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 $p, q :: CTree_\bot\ e\ a$ and $f :: a \to b$ such that $x \sqsubseteq y$ implies $x = y$ for all $x, y :: b$. If p is safe and $p\ {}_\bot\!\lesssim\ q$, then $fmap\ f\ p \cong fmap\ f\ q$.*

Proof. By definition $fmap\ f\ p = p \gg\!= \lambda x \to return\ (f\ x)$. By reflexivity, we have $return\ (f\ x)\ {}_\bot\!\lesssim\ return\ (f\ x)$ for all $x :: a$ and thus by the congruence law for $\gg\!=$ we have

$$fmap\ f\ p = p \gg\!= f\ {}_\bot\!\lesssim\ q \gg\!= f = fmap\ f\ q$$

Since the ordering on the type b is trivial, we obtain by Proposition 2 that $fmap\ p\ {}_\bot\!\cong\ fmap\ f\ q$. In turn, since p and thus $fmap\ f\ p$ is safe, we obtain by Proposition 1 $fmap\ f\ p \cong fmap\ f\ q$. □

Similar to 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, we reconsider the expression language from section 3, but without conditionals as these are not necessary here to illustrate the calculation technique:

data $Expr = Val\ Int \mid Add\ Expr\ Expr \mid Print\ Expr$

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

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 $CTree_\bot\ PrintEff\ Int$ instead of just $CTree\ PrintEff\ Int$ since (ordered) skew bisimilarity is only defined on partial choice trees.

We aim to calculate a compiler $compile :: Expr \to Code$ that targets a register machine $exec :: Code \to Config \to CTree_\bot\ PrintEff\ Config$ where

type $Conf = (Acc, Mem\ Int)$
type $Acc = Int$

The compiler correctness property is essentially the same as specification (10) but is now formulated using ordered skew bisimilarity:

$$\textbf{do}\ v \leftarrow eval\ e;\ return\ (v, empty)\ {}_\bot\!\lesssim\ exec\ (compile\ e)\ (a, empty) \tag{11}$$

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 we may start with any memory m (not just the empty memory) as long as m is free from the register r onwards:

$$\mathit{freeFrom}\ r\ m \ \Rightarrow\ \mathbf{do}\ v \leftarrow \mathit{eval}\ e\,;\ \mathit{exec}\ c\ (v, m)\ {}_{\perp}\!\lesssim\ \mathit{exec}\ (\mathit{comp}\ e\ r\ c)\ (a, m) \tag{12}$$

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

$$s \sqsubseteq s' \ \Rightarrow\ \mathit{exec}\ c\ s\ {}_{\perp}\!\lesssim\ \mathit{exec}\ c\ s' \qquad (\mathit{exec}\text{-mono})$$

Intuitively, the monotonicity property for *exec* means that the machine cannot observe whether a reading a register has failed or not. This property holds by construction since the two operations on memory configurations – *set* and *get* – are monotone. Calculations rely heavily on the *exec*-mono property.

We now calculate the definition of *comp* from the specification (12) by structural induction on e. That is, we may assume some memory m and register r with $\mathit{freeFrom}\ r\ m$ and then aim to manipulate the left-hand side of the inequality into a term of the form $\mathit{exec}\ c'\ (a, m)$, which can be made to match the right-hand side of the inequality by defining $\mathit{comp}\ e\ r\ c = c'$ for that case of e.

The case for $e = \mathit{Val}\ n$ is straightforward as it is very similar to the calculation for a stack machine in section 3:

$$\begin{array}{ll}
 & \mathbf{do}\ v \leftarrow \mathit{eval}\ (\mathit{Val}\ n)\,;\ \mathit{exec}\ c\ (v, m) \\
= & \quad \{\text{definition of } \mathit{eval}\} \\
 & \mathbf{do}\ v \leftarrow \mathit{return}\ n\,;\ \mathit{exec}\ c\ (v, m) \\
\cong & \quad \{\text{monad laws}\} \\
 & \mathit{exec}\ c\ (n, m) \\
= & \quad \{\text{define: } \mathit{exec}\ (\mathit{LOAD}\ n\ c)\ (a, m) = \mathit{exec}\ c\ (n, m)\} \\
 & \mathit{exec}\ (\mathit{LOAD}\ x\ c)\ (a, m)
\end{array}$$

In the last step, similarly to the calculation for a stack machine, we aim to arrive at a term of the form $\mathit{exec}\ c'\ (a, m)$. That is, we must solve the inequality

$$\mathit{exec}\ c'\ (a, m)\ \ \gtrsim_{\perp}\ \mathit{exec}\ c\ (n, m)$$

which we achieve by first strengthening it to an equation

$$\mathit{exec}\ c'\ (a, m) = \mathit{exec}\ c\ (n, m)$$

and then solving the equation by instantiating c' with $\mathit{LOAD}\ n\ c$ so that we can take the equation as a clause of the definition of *exec*.

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

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

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

$$\mathbf{do}\ n_2 \leftarrow \mathit{eval}\ y\,;\ \mathit{exec}\ c'\ (n_2, m')$$

Given the term we currently have, this means that we must solve the inequality

$$\mathbf{do}\ n_2 \leftarrow eval\ y;\ exec\ c\ (n_1 + n_2, m)\ {}_{\bot}{\lesssim}\ \mathbf{do}\ n_2 \leftarrow eval\ y;\ exec\ c'\ (n_2, m')$$

Recall that in the calculation for stack machines, solving this kind of inequality was achieved by introducing an instruction that 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 right-hand side, n_2 is already in the accumulation register. It thus remains to pick a suitable m' which 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:

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

To obtain our desired subterm to apply the induction hypothesis, it thus remains to solve the inequality:

$$exec\ c\ (n_1 + n_2, set\ r\ n_1\ m)\ {}_{\bot}{\lesssim}\ exec\ c'\ (n_2, m')$$

This can be achieved by strengthening this inequality to an equation

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

and then solving this equation by instantiating $m' = set\ r\ n_1\ m$ 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*. To achieve that we, first rewrite the right-hand side into an equivalent form by appealing to the *set-get* law and the monad laws:

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

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

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

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

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

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

After clearing this hurdle, the calculation continues to the final goal in straightforward fashion:

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

$$
\begin{aligned}
{}_{\bot}{\lesssim}\quad & \{\, \text{induction hypothesis for } x \text{ and } r \,\} \\
& \mathit{exec}\ (\mathit{comp}\ x\ r\ (\mathit{STORE}\ r\ (\mathit{comp}\ y\ (\mathit{next}\ r)\ (\mathit{ADD}\ r\ c))))\ (a, m)
\end{aligned}
$$

Finally, we conclude the compiler calculation with the case for $e = \mathit{Print}\ x$. 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 easily achieved simply by introducing an instruction:

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

This concludes the calculation of *comp* and *exec*. The top-level compiler *compile* can then be calculated using specification (11):

$$
\begin{aligned}
& \mathbf{do}\ v \leftarrow \mathit{eval}\ e\,;\ \mathit{return}\ (v, \mathit{empty}) \\
= \quad & \{\, \text{define: } \mathit{exec}\ \mathit{HALT}\ s = \mathit{return}\ s \,\} \\
& \mathbf{do}\ v \leftarrow \mathit{eval}\ e\,;\ \mathit{exec}\ \mathit{HALT}\ (v, \mathit{empty}) \\
{}_{\bot}{\lesssim} \quad & \{\, \text{specification (12) \& } \mathit{first\text{-}empty} \,\} \\
& \mathit{exec}\ (\mathit{comp}\ e\ \mathit{first}\ \mathit{HALT})\ (v, \mathit{empty})
\end{aligned}
$$

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 5.

Finally, using the fact that *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:

$$\mathit{eval}\ e \cong \mathbf{do}\ (a, m) \leftarrow \mathit{exec}\ (\mathit{compile}\ e)\ (0, \mathit{empty})\,;\ \mathit{return}\ a$$

That is evaluating the expression e is the same as first compiling e, then running the resulting code on the register 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 (11) we have just proved:

$$\mathbf{do}\ v \leftarrow \mathit{eval}\ e\,;\ \mathit{return}\ (v, \mathit{empty})\ {}_{\bot}{\lesssim}\ \mathit{exec}\ (\mathit{compile}\ e)\ (0, \mathit{empty})$$

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

$$\mathit{fmap}\ \mathit{fst}\ (\mathbf{do}\ v \leftarrow \mathit{eval}\ e\,;\ \mathit{return}\ (v, \mathit{empty})) \cong \mathit{fmap}\ \mathit{fst}\ (\mathit{exec}\ (\mathit{compile}\ e)\ (0, \mathit{empty}))$$

We can then apply the monad laws to simplify this to

$$\mathit{eval}\ e \cong \mathbf{do}\ (a, m) \leftarrow \mathit{exec}\ (\mathit{compile}\ e)\ (0, \mathit{empty})\,;\ \mathit{return}\ a$$

We have demonstrated the calculation technique on a minimal example: a simple expression language with a single algebraic effect. But the technique scales to more realistic source languages as well. The extensions illustrated in section 4.3 also apply to ordered skew bisimilarity, which allows us to calculate register machine compilers for source languages featuring non-termination and concurrency. Among other examples, we have calculated a register machine compiler for a

Target language

```
data Code = LOAD Int Code | STORE Reg Code | ADD Reg Code | PRINT Code
```

Compiler

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

comp :: Expr → Reg → Code → Code
comp (Val n)     r c = LOAD 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)
```

Virtual machine

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

exec :: Code → Conf → CTree⊥ PrintEff Conf
exec (LOAD 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 m r; exec c (b + a, m)
exec (PRINT c)   (a, m) = do print a; exec c (a, m)
exec HALT         s      = return s
```

Fig. 5. Compiler and register machine for the simple expression language extended with print.

lambda calculus with algebraic effects and concurrency. The formalisation of these calculations can be found in the supplementary material.

6 RELATED WORK

6.1 Partial bisimilarity

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 as safe and weakens bisimilarity for all unsafe transition labels. In this article we considered the case where only $\uparrow$ *Inl Stuck* is considered unsafe, but this restriction is not necessary. Indeed, the Agda formalisation in the supplementary material is parametrised by the set of labels that is considered unsafe. This simplifies the formalisation, since it means that bisimilarity is just 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 $s \xRightarrow{l} s'$ implies $t \xRightarrow{l} t'$ and $s' \;{}_U\!\sim\; t'$, and
(2) $t \xRightarrow{l} t'$ implies $s \xRightarrow{l} s'$ and $s' \;{}_U\!\sim\; t'$.

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

However, this notion of partiality is not suitable for our purposes. If we instantiate $U = \{l \mid l \neq\uparrow \mathit{Inl}\ \mathit{Stuck}\}$, 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.

6.2 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 component techniques have been in turn introduced by Reynolds [1972] a decade earlier and have proved essential also in later correct-by-construction compiler derivation techniques proposed by Meijer [1992], Ager et al. [2003], and Bahr and Hutton [2015].

6.3 Partial specification

As mentioned in section 2.1, partial specifications of compiler correctness have long been adopted by compiler verification projects [Leroy 2009]. However, we are not aware of a compiler calculation technique that employs partial specifications.

An alternative approach to the use of partial specifications is to rule out unsafe source language programs altogether. As Pickard and Hutton [2021] have demonstrated, by defining the syntax and semantics of the source language in a dependently typed meta language – so-called *intrinsic typing* – we can encode the source language's type system so that the specification and the calculation only has to deal with safe behaviours. Unsafe programs are simply not expressible, and we thus do not even have to consider them in the definition of *eval*. However, so far this has only been applied to simple languages without any side effects. Moreover, encoding the type system in the meta language can be tricky and coming up with a corresponding type system for the target language can be even more difficult as we need to anticipate how the target machine is going to work.

The use of skew bisimilarity circumvents these shortcomings entirely, allowing us to calculate compilers for complex languages with sophisticated type systems. To put this claim to the test, we have calculated a compiler for Simply RaTT [Bahr et al. 2019], a higher-order functional reactive programming language that features modal type operators and a Fitch-style type system [Clouston 2018]. Calculating a compiler for such a language using the dependently typed approach of Pickard and Hutton [2021] would be unfeasible: The proof of type soundness for this language, i.e. that well-typed programs only exhibit safe behaviours, requires a complex 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.

7 CONCLUSION AND FURTHER WORK

With skew bisimilarity, we have introduced a weaker form of bisimilarity that can simplify reasoning about unsafe source languages and allows us to derive compilers that can produce more efficient code. In addition, it 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. However, 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. 2023] may provide the semantic foundation that enables calculation technique for closing that gap.

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, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg. 2019. Simply RaTT: A Fitch-style Modal Calculus for Reactive Programming Without Space Leaks. *Proc. ACM Program. Lang.* 3, ICFP (July 2019), 109:1–109:27. https://doi.org/10.1145/3341713 00000.

Patrick Bahr and Graham Hutton. 2015. Calculating Correct Compilers. *Journal of Functional Programming* 25 (2015).

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

Patrick Bahr and Graham Hutton. 2022. Monadic Compiler Calculation. *Proceedings of the ACM on Programming Languages* 6, ICFP (2022).

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

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

Nicolas Chappe, Paul He, Ludovic Henrio, Yannick Zakowski, and Steve Zdancewic. 2023. Choice Trees: Representing Nondeterministic, Recursive, and Impure Programs in Coq. *Proceedings of the ACM on Programming Languages* 4, POPL (2023).

Ranald Clouston. 2018. Fitch-Style Modal Lambda Calculi. In *Foundations of Software Science and Computation Structures (Lecture Notes in Computer Science)*, Christel Baier and Ugo Dal Lago (Eds.). Springer International Publishing, 258–275. 00006.

Nils Anders Danielsson and Thorsten Altenkirch. 2010. Subtyping, Declaratively. In *Proceedings of the International Conference on Mathematics of Program Construction.* Vol. 6120. Lecture Notes in Computer Science.

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* (San Diego, California, USA).

Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Communications of the ACM* 52, 7 (2009).

John McCarthy and James Painter. 1967. Correctness of a Compiler for Arithmetic Expressions. In *Mathematical Aspects of Computer Science (Proceedings of Symposia in Applied Mathematics, Vol. 19).* American Mathematical Society.

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

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

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

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

J. J. M. M. Rutten. 2000. Coalgebra, Concurrency, and Control. In *Discrete Event Systems: Analysis and Control*, R. Boel and G. Stremersch (Eds.). Springer US, Boston, MA, 31–38. https://doi.org/10.1007/978-1-4615-4493-7_2

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

Janis Voigtländer. 2008. Asymptotic Improvement of Computations over Free Monads. In *Proceedings of the International Conference on Mathematics of Program Construction.*

Mitchell Wand. 1982. Deriving Target Code as a Representation of Continuation Semantics. *ACM Transactions on Programming Languages and Systems* 4, 3 (1982), 22 pages. http://doi.acm.org/10.1145/357172.357179