Monadic Compiler Calculation (Functional Pearl)

PATRICK BAHR, IT University of Copenhagen, Denmark
GRAHAM HUTTON, University of Nottingham, UK

Bahr and Hutton recently developed a new approach to calculating correct compilers directly from specifications of their correctness. However, the methodology only considers *converging* behaviour of the source language, which means that the compiler could potentially produce arbitrary, erroneous code for source programs that diverge. In this article, we show how the methodology can naturally be extended to support the calculation of compilers that address both convergent and divergent behaviour *simultaneously*, without the need for separate reasoning for each aspect. Our approach is based on the use of the partiality monad to make divergence explicit, together with the use of strong bisimilarity to support equational-style calculations, but also generalises to other forms of effect by changing the underlying monad.

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

Additional Key Words and Phrases: program calculation, bisimilarity, divergence, non-determinism

ACM Reference Format:
Patrick Bahr and Graham Hutton. 2022. Monadic Compiler Calculation (Functional Pearl). *Proc. ACM Program. Lang.* 6, ICFP, Article 93 (August 2022), 29 pages. https://doi.org/10.1145/3547624

93

1 INTRODUCTION

The aim of *program calculation* is to derive correct-by-construction programs from specifications of their desired behaviour [Backhouse 2003]. For example, program calculation techniques can be used to derive compilers from specifications of their correctness. This approach allows us to systematically *discover* compilation techniques, while at the same time obtaining *proofs* that they are correct. The starting point is a semantics for the compiler's source and target languages, along with a formal statement that the as-yet undefined compiler preserves the semantics of programs. We then proceed to prove the correctness property by calculation, in the process of which the definition of the compiler is discovered case by case. In addition, the target language and its semantics may also be undefined, and then derived by calculation at the same time.

With existing compiler calculation techniques [Ager et al. 2003; Bahr and Hutton 2015; Meijer 1992; Reynolds 1972; Sestoft 1997; Wand 1982], the semantics of the source language is given by an inductive big-step semantics, or a (structurally) recursive definitional interpreter. As such, the semantics do not capture non-terminating behaviour, and calculation is limited to inductive reasoning principles that cannot account for non-termination. Hence, these techniques are unsound for non-total languages. In particular, if a source program diverges, then the compiler correctness specification makes no guarantees about the behaviour of the resulting target program.

In compiler *verification*, reasoning about divergence is a long-solved problem: big-step semantics may be defined coinductively [Leroy 2006a], and definitional interpreters may be defined by general

Authors' addresses: Patrick Bahr, IT University of Copenhagen, Denmark; Graham Hutton, University of Nottingham, UK.

2475-1421/2022/8-ART93
https://doi.org/10.1145/3547624

recursion using the partiality monad [Capretta 2005; Danielsson 2012]. Unfortunately, the reasoning principles used in these settings are incompatible with compiler *calculation*, because they are based on weak (bi)similarity. Informally, two computations are weakly bisimilar iff they have the same behaviour modulo logical (silent) computation steps. In compiler correctness, we seek to establish a weak bisimilarity between the semantics of the source and target programs. In this setting, logical steps are ignored as they are artefacts of the way the semantics are formulated, and in general we cannot expect that the source and target semantics align in the use of such steps. However, it is well-known that one cannot combine equational-style proofs of weak bisimilarity with a coinduction reasoning principle [Danielsson 2012], as we discuss further in section 2.

Instead of *weak* bisimilarity, we propose using *strong* bisimilarity as the underlying reasoning principle for compiler calculation. While unsuitable for compiler verification, strong bisimilarity fits the calculational approach as it supports equational-style coinduction proofs. Conversely, calculation eliminates the otherwise fatal drawback of strong bisimilarity: because the semantics of the target language can also be derived, the correct number of logical steps can be inserted in the target semantics so that it aligns with the source semantics. Moreover, as with the rest of the derived components, where to add these steps naturally falls out of the calculation process.

In this article, we extend the compiler calculation techniques of Bahr and Hutton [2015] to use strong bisimilarity, by defining the semantics of source and target language in terms of the partiality monad in a similar manner to Danielsson [2012]. Crucially, however, we use a reasoning principle based on strong rather than weak bisimilarity, which allows us to derive compilers directly from their specifications, as opposed to verifying existing compilers. In addition, we argue that the resulting calculations are simpler than the typical coinduction proofs used in verifications. In particular, our calculations inherit the simplicity of previous compiler calculation methods, as we can reason about non-termination using equational monadic laws.

We demonstrate our new compiler calculation technique on three examples. First of all, we use a minimal language with a looping primitive to illustrate the shortcomings of previous techniques, and how these can be remedied using the partiality monad and strong bisimilarity (section 2). Secondly, to demonstrate that the methodology scales to more realistic languages, we apply it to a call-by-value version of the lambda calculus (section 3). To the best of our knowledge, this is the first calculation of a compiler for the untyped lambda calculus that establishes full correctness, i.e. taking account of both convergent and divergent behaviour. Finally, we show that the methodology can be applied to effects other than divergence by replacing the partiality monad with a different monad. To this end, we calculate a compiler for a language with exceptions and interrupts (section 4), which uses a non-determinism monad to take account of the fact that programs in this language may have more than one possible result value. The aim of these examples is not to develop verified real-world compilers, but rather to demonstrate the utility of our methodology for deriving sound compilation techniques in the presence of non-termination and other effects.

The article is aimed at readers with some basic experience of formal semantics and reasoning, but we do not require previous knowledge of compiler calculation. We use Haskell notation as our meta-language for accessibility, but assume that the language is total. Whereas in many articles calculations are often omitted or compressed for brevity, here they are the central focus, so they are generally presented in detail. All the calculations have been mechanically checked in Agda, and the source code for these calculations, together with a number of additional compiler calculations, are available as online supplementary material for the article [Bahr and Hutton 2022].

2 A SIMPLE NON-TOTAL LANGUAGE

To introduce our new methodology, in this section we consider a minimal source language that comprises arithmetic expressions that are built up from integer values using an addition operator, extended with a primitive that simply loops forever without producing any result value:

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

We begin by reviewing the compiler calculation approach of Bahr and Hutton [2015], then identify its shortcoming for non-total languages, and afterwards gradually refine the compiler specification into its final form (Theorem 2.1) suitable for calculating a compiler for the above language.

2.1 Inductive Specification

The semantics for expressions can naturally be captured by a definitional interpreter [Reynolds 1972] that evaluates an expression to an integer value. Importantly, the semantics is not total because it may enter an infinite loop and hence never produce a result value; for this initial version of the semantics we allow ourselves to use the standard, non-total version of Haskell:

$$
\begin{aligned}
&eval :: Expr \to Int\\
&eval\ (Val\ n) \quad\ \ = n\\
&eval\ (Add\ x\ y) = eval\ x + eval\ y\\
&eval\ Loop \qquad\ = eval\ Loop
\end{aligned}
$$

Our goal now is to calculate a compiler $comp :: Expr \to Code$ that translates an expression into code for our (as of yet unspecified) target language. We assume that the compiler targets a stack-based machine, whose semantics is given by a function $exec :: Code \to Stack \to Stack$, where $\textbf{type}\ Stack = [Int]$ is the stack type for the machine. The definitions for $Code$ and $exec$ are not given up front, but rather will fall naturally out of the calculation of the compiler. Because $exec$ defines the semantics of a virtual machine, we expect it to be a tail-recursive function.

Prior to specifying the desired behaviour of the compiler, we generalise the function $comp$ to take additional code to be executed after the compiled code. The addition of such a *code continuation* is a key aspect of the underlying methodology and significantly simplifies the resulting calculations. Moreover, the use of code continuations makes explicit that code can be composed sequentially but is not necessarily linear. Using this idea, our goal now is to establish the following compiler correctness property for the generalised compilation function $comp :: Expr \to Code \to Code$:

$$exec\ c\ (eval\ e : s) = exec\ (comp\ e\ c)\ s$$

That is, compiling an expression and then executing the resulting code together with additional code gives the same result as executing the additional code with the value of the expression on top of the stack. The proof of the compiler correctness property proceeds by induction on the structure of the source expression e. For each case of e, we start on the left-hand side of the equation, i.e. the term $exec\ c\ (eval\ e : s)$, and seek to transform it by equational reasoning into a term of the form $exec\ c'\ s$ for some code c'. We then define $comp\ e\ c = c'$, which gives us a definition of the compiler in this case that satisfies the correctness property above.

For example, in the case for $Val\ n$ we first apply the definition of the evaluation function:

$$
\begin{aligned}
&\quad exec\ c\ (eval\ (Val\ n) : s)\\
= &\quad \{\ \text{definition of } eval\ \}\\
&\quad exec\ c\ (n : s)
\end{aligned}
$$

Then, to complete the calculation, we need to arrive at a term of the form *exec* c' s. That is, we have to find some code c' that solves the following equation:

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

Note that we cannot simply use this equation as a defining clause for *exec*, as n and c would be unbound in the body of the definition. The solution is to package these two variables up in the code argument c', which can freely be instantiated as it is existentially quantified, whereas all the other variables in the equation are universally quantified. This can be achieved by adding a new constructor to the *Code* datatype that takes n and c as arguments,

$PUSH :: Int \to Code \to Code$

and defining a new clause for the function *exec* as follows:

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

That is, the code $PUSH\ n\ c$ is executed by pushing the integer n onto the top of the stack and then executing the remaining code c, which motivates the name for the new code constructor. This definition solves the above equation, and allows us to conclude the calculation:

$$\begin{aligned} & exec\ c\ (n : s) \\ = \quad & \{\ \text{definition of } exec\ \} \\ & exec\ (PUSH\ n\ c)\ s \end{aligned}$$

In summary, we have discovered initial cases for both the stack machine and the compiler:

data $Code = PUSH\ Int\ Code$

$exec :: Code \to Stack \to Stack$
$exec\ (PUSH\ n\ c)\ s = exec\ c\ (n : s)$

$comp :: Expr \to Code \to Code$
$comp\ (Val\ n)\ c = PUSH\ n\ c$

The calculations for the remaining two cases, $Add\ x\ y$ and *Loop*, should complete these definitions and the compiler correctness proof. The case for addition does not present any problems. However, for *Loop*, applying the evaluation function brings us straight back to the same term:

$$\begin{aligned} & exec\ c\ (eval\ Loop : s) \\ = \quad & \{\ \text{definition of } eval\ \} \\ & exec\ c\ (eval\ Loop : s) \end{aligned}$$

During such a calculation we aim to simplify the source expression in order to then apply an induction hypothesis. However, we cannot apply an induction hypothesis here, because *Loop* is not smaller than the expression we started out with, namely *Loop* itself. Fundamentally, the problem is that the evaluation function *eval* is not compositional, i.e. structurally recursive, because in the case for *Loop* we make a recursive call on precisely the same expression.

Bahr and Hutton [2015] addressed this problem by rephrasing the non-compositional evaluation function as a big-step operational semantics $e \Downarrow v$, and then performing induction on the structure of the operational semantics, rather than on the structure of the source language. In turn, the compiler correctness property is rephrased as follows:

$$e \Downarrow v \quad \Rightarrow \quad exec\ c\ (v : s) = exec\ (comp\ e\ c)\ s$$

However, due to the evaluation pre-condition, this property now only captures *converging* behaviour of the source language. In particular, it makes no stipulations about how the compiled code should behave if the source expression *diverges*, i.e. does not terminate. As such, the code produced by the compiler for the non-terminating expression *Loop* could be entirely arbitrary.

2.2 Coinductive Specification

Rather than defining the semantics of the source language inductively, we can use a coinductive definition to capture both the convergent and divergent behaviour. To this end we will follow Danielsson's [2012] approach and use Capretta's *partiality monad* [2005]:

$$\mathbf{codata}\ \mathit{Partial}\ a = \mathit{Now}\ a \mid \mathit{Later}\ (\mathit{Partial}\ a)$$

Here we write **codata** to indicate that the type is coinductively defined, i.e. defined as a greatest fixed point. The idea is that *Now* returns a value immediately, while *Later* postpones a computation by one time step. The partiality type forms a monad, with the operations defined as follows:

$$\begin{array}{l}
\mathit{return} :: a \to \mathit{Partial}\ a \\
\mathit{return}\ x = \mathit{Now}\ x
\end{array}$$

$$\begin{array}{l}
(\ggg\!=) :: \mathit{Partial}\ a \to (a \to \mathit{Partial}\ b) \to \mathit{Partial}\ b \\
\mathit{Now}\ x \ \ggg\!= f = f\ x \\
\mathit{Later}\ p \ggg\!= f = \mathit{Later}\ (p \ggg\!= f)
\end{array}$$

We'll consider the monad laws later in this section once we have defined a suitable notion of bisimilarity. In addition, we can define a computation that never terminates:

$$\begin{array}{l}
\mathit{never} :: \mathit{Partial}\ a \\
\mathit{never} = \mathit{Later}\ \mathit{never}
\end{array}$$

We can now rewrite our semantics as a corecursively defined function of type $\mathit{Expr} \to \mathit{Partial}\ \mathit{Int}$. We ensure that the definition is productive by making every recursive call either structurally recursive, as in the case for addition, or *guarding* it with a *Later*, as in the case for loop:

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{eval} :: \mathit{Expr} \to \mathit{Partial}\ \mathit{Int}} \\
\mathit{eval}\ (\mathit{Val}\ n) & = & \mathit{return}\ n \\
\mathit{eval}\ (\mathit{Add}\ x\ y) & = & \mathbf{do}\ m \leftarrow \mathit{eval}\ x \\
 & & \phantom{\mathbf{do}\ } n \leftarrow \mathit{eval}\ y \\
 & & \phantom{\mathbf{do}\ } \mathit{return}\ (m + n) \\
\mathit{eval}\ \mathit{Loop} & = & \mathit{Later}\ (\mathit{eval}\ \mathit{Loop})
\end{array}$$

Compared to the previous interpreter, our corecursive version of *eval* is total at the level of the meta-language. That is, instead of relying on a non-total meta language, we have captured the divergent behaviour explicitly using the partiality monad. As such, we can formally reason about non-termination and thus prove that non-termination is preserved by the compiler. On a practical level, this also means that we can use the definition above in a total language such as Agda.

Since our goal is to calculate a compiler that preserves all behaviours of the source language, including non-termination, we also need that the target machine can diverge:

$$\mathit{exec} :: \mathit{Code} \to \mathit{Stack} \to \mathit{Partial}\ \mathit{Stack}$$

Adapting the compiler correctness property to account for the use of the partiality monad in both the source and target languages only requires a minor notational change:

$$\begin{array}{l}
\mathbf{do}\ v \leftarrow \mathit{eval}\ e \\
\phantom{\mathbf{do}\ } \mathit{exec}\ c\ (v : s)
\end{array} \quad = \quad \mathit{exec}\ (\mathit{comp}\ e\ c)\ s$$

Note that both sides of the equality are terms of type *Partial Stack*, a coinductively defined type. Depending on the nature of the meta language, simple equality may be too strict, i.e. the reasoning principles afforded by the notion of equality might be too weak to actually prove this property. Instead, we need a notion of *bisimilarity* [Park 1981] that supports a coinductive reasoning principle and a calculation-style proof, i.e. equalities that can be chained together by transitivity. In the next section we explore what a suitable notion of bisimilarity should look like.

2.3 Bisimilarity

In the literature on compiler verification (see section 5), one typically finds a form of *weak* bisimilarity. Intuitively, this notion expresses that one term converges iff another term converges, but it does not matter how many *steps* they take to converge, i.e. how many times we encounter a *Later*. To formally define this notion of weak bisimilarity, we first define when a computation $p :: \mathit{Partial}\ a$ converges to a value $v :: a$ by an inductively defined relation $p \Downarrow v$:

$$\frac{}{\mathit{Now}\ v \Downarrow v} \qquad\qquad \frac{p \Downarrow v}{\mathit{Later}\ p \Downarrow v}$$

That is, *Now* v immediately converges to v, while *Later* p converges to a value if p converges to this value. Given two computations $p, q :: \mathit{Partial}\ a$ we say that p and q are weakly bisimilar, written as $p \approx q$, if they coincide in terms of their convergence behaviours:

$$p \Downarrow v \quad \text{iff} \quad q \Downarrow v \qquad \text{for all } v$$

The notion of weak bisimilarity abstracts away from how many steps are required. This makes it suitable for use with our compiler correctness property, because executing the compiled code for an expression may take a different number of steps compared to evaluating the expression:

$$\begin{array}{l}\mathbf{do}\ v \leftarrow \mathit{eval}\ e \\ \quad\ \mathit{exec}\ c\ (v : s)\end{array} \quad \approx \quad \mathit{exec}\ (\mathit{comp}\ e\ c)\ s$$

Unfortunately, weak bisimilarity does not have a coinductive reasoning principle that is compatible with a calculational style. In particular, calculation relies on the use of transitivity to chain together successive reasoning steps. If we assumed such a coinductive reasoning principle, we could prove $p \approx q$ for any $p, q : \mathit{Partial}\ a$ by the following coinductive argument:

$$\begin{array}{ll}
 & p \\
\approx & \{\ p \text{ and } \mathit{Later}\ p \text{ only differ in the number of steps}\ \} \\
 & \mathit{Later}\ p \\
\approx & \{\ \text{coinductive hypothesis } p \approx q \text{, guarded by } \mathit{Later}\ \} \\
 & \mathit{Later}\ q \\
\approx & \{\ q \text{ and } \mathit{Later}\ q \text{ only differ in the number of steps}\ \} \\
 & q
\end{array}$$

To avoid this problem, we will use the stricter notion of *strong bisimilarity*, or just *bisimilarity* for short. To this end, we first define a step-indexed version of the convergence relation, which counts the number of steps, i.e. uses of *Later*, that are required:

$$\frac{}{\mathit{Now}\ v \Downarrow_i v} \qquad\qquad \frac{p \Downarrow_i v}{\mathit{Later}\ p \Downarrow_{i+1} v}$$

We can think of $\Downarrow_i$ as capturing the idea of convergence using i units of 'fuel'; the base case for *Now* v uses an arbitrary index i rather than zero because we don't need to use all the fuel that is provided. Given two computations $p, q :: \mathit{Partial}\ a$ we say that p and q are bisimilar, written as $p \cong q$, if they coincide in terms of their step-counting convergence behaviours:

$$p \Downarrow_i v \quad \text{iff} \quad q \Downarrow_i v \qquad \text{for all } v \text{ and } i$$

2.4 Compiler Correctness

Using the above notion of bisimilarity, we can now formulate the final version of the compiler correctness property for our simple language of expressions:

Theorem 2.1 (compiler correctness).

$$\begin{array}{l}\textbf{do}\ v \leftarrow \mathit{eval}\ e \\ \quad \mathit{exec}\ c\ (v : s)\end{array} \quad \cong \quad \mathit{exec}\ (\mathit{comp}\ e\ c)\ s$$

This property is much stricter than the version using weak bisimilarity. In general this is a problem, as we expect compiled code to take a different number of steps to execute compared to the evaluation of the original source expression. However, these 'steps' are not actual computation steps, but rather 'logical steps', in the form of uses of *Later* that have been inserted to ensure a well-defined evaluation function *eval*. Together with the fact that *exec* is not given up-front but is instead calculated, this leaves us the freedom to define *exec* so that it takes just the right number of logical steps. Moreover, as we will see, where to place these logical steps will fall out of the calculation process, in the same way that the rest of the definition of the *exec* function.

The benefit that bisimilarity gives us over weak bisimilarity is a coinductive reasoning principle that is compatible with transitive reasoning. To this end, we define a notion of step-indexed bisimilarity. Given two computations $p, q :: \mathit{Partial}\ a$ and a natural number i, we say that p and q are i-bisimilar, written as $p \cong_i q$, if the following condition holds:

$$p \Downarrow_j v \quad \text{iff} \quad q \Downarrow_j v \qquad \text{for all } v \text{ and } j < i$$

The relation $\cong_i$ is explicitly defined to be downwards closed, i.e. $p \cong_i q$ implies $p \cong_j q$ for all $j \leq i$. This property is crucial to ensure that $\cong_i$ is a congruence for the monadic bind operator. Moreover, using this definition, we have $p \cong q$ iff $p \cong_i q$ for all step counts i. Hence, our compiler correctness theorem can be established by proving the following by induction on both i and e:

$$\begin{array}{l}\textbf{do}\ v \leftarrow \mathit{eval}\ e \\ \quad \mathit{exec}\ c\ (v : s)\end{array} \quad \cong_i \quad \mathit{exec}\ (\mathit{comp}\ e\ c)\ s \tag{1}$$

During such a proof, we can assume that for all step counts $j < i$, we have:

$$\begin{array}{l}\textbf{do}\ v \leftarrow \mathit{eval}\ e' \\ \quad \mathit{exec}\ c'\ (v : s')\end{array} \quad \cong_j \quad \mathit{exec}\ (\mathit{comp}\ e'\ c')\ s'$$

This inductive hypothesis can then be used by applying the following proof rule:

$$\frac{p \cong_j q \quad \text{for all } j < i}{\mathit{Later}\ p \cong_i \mathit{Later}\ q} \tag{2}$$

Because we perform induction on the expression at the same time, we can also assume the following induction hypothesis for all expressions e' that are structurally smaller than e:

$$\begin{array}{l}\textbf{do}\ v \leftarrow \mathit{eval}\ e' \\ \quad \mathit{exec}\ c'\ (v : s')\end{array} \quad \cong_i \quad \mathit{exec}\ (\mathit{comp}\ e'\ c')\ s'$$

In addition, we will use the fact that *Partial* satisfies the monad laws up to bisimilarity and therefore the monadic laws also hold for our notion of i-bisimilarity:

$$\begin{array}{lcl} \mathit{return}\ x \gg\!= f & \cong & f\ x \\ \mathit{mx} \gg\!= \mathit{return} & \cong & \mathit{mx} \\ (\mathit{mx} \gg\!= f) \gg\!= g & \cong & \mathit{mx} \gg\!= (\lambda x \rightarrow (f\ x \gg\!= g)) \end{array}$$

2.5 Compiler Calculation

We prove property (1) by induction on the step count i and expression e, from which compiler correctness (Theorem 2.1) follows immediately. For each case of e, we start on the left-hand side of the property and seek to transform it into the form $\mathit{exec}\ c'\ s$ for some code c'. We can then set $\mathit{comp}\ e\ c = c'$ as the definition of the compiler in this case. During the calculation for each case, we also discover a new clause for the definition of the virtual machine *exec*, driven by the desire to transform the term being manipulated into the required form.

The cases for values and addition proceed in the same manner as [Bahr and Hutton 2015], except that because we are now working in a monadic setting we also use the monad laws:

Case: $e = \mathit{Val}\ n$

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

In the final step above, we introduce a code constructor *PUSH* and a corresponding clause for *exec*. These definitions follow from the fact that we are aiming for a term of the form $\mathit{exec}\ c'\ s$ and thus need to solve the equation $\mathit{exec}\ c'\ s \cong_i \mathit{exec}\ c\ (n : s)$. We solve this equation by strengthening i-bisimilarity to equality, i.e. $\mathit{exec}\ c'\ s = \mathit{exec}\ c\ (n : s)$, and instantiating c' to $\mathit{PUSH}\ n\ c$ so that the equation becomes a clause for *exec*. All the other steps above are entirely mechanical.

For the *Add* case, our goal is to apply the induction hypothesis for the argument expressions x and y in order to ultimately obtain a term of the required form $\mathit{exec}\ c'\ s$:

Case: $e = \mathit{Add}\ x\ y$

$$\begin{array}{ll}
& \mathbf{do}\ v \leftarrow \mathit{eval}\ (\mathit{Add}\ x\ y) \\
& \quad\ \mathit{exec}\ c\ (v : s) \\
\cong_i & \{ \text{definition of } \mathit{eval} \} \\
& \mathbf{do}\ v \leftarrow \mathbf{do}\ m \leftarrow \mathit{eval}\ x \\
& \qquad\qquad\quad\ n \leftarrow \mathit{eval}\ y \\
& \qquad\qquad\quad\ \mathit{return}\ (m + n) \\
& \quad\ \mathit{exec}\ c\ (v : s) \\
\cong_i & \{ \text{monad laws} \} \\
& \mathbf{do}\ m \leftarrow \mathit{eval}\ x \\
& \quad\ n \leftarrow \mathit{eval}\ y
\end{array}$$

$$
\begin{array}{ll}
& \quad exec\ c\ ((m+n) : s) \\
\cong_i & \{\ \text{define: } exec\ (ADD\ c)\ (n : m : s) = exec\ c\ ((m+n) : s)\ \} \\
& \mathbf{do}\ m \leftarrow eval\ x \\
& \quad n \leftarrow eval\ y \\
& \quad exec\ (ADD\ c)\ (n : m : s) \\
\cong_i & \{\ \text{induction hypothesis for } y\ \} \\
& \mathbf{do}\ m \leftarrow eval\ x \\
& \quad exec\ (comp\ y\ (ADD\ c))\ (m : s) \\
\cong_i & \{\ \text{induction hypothesis for } x\ \} \\
& exec\ (comp\ x\ (comp\ y\ (ADD\ c)))\ s
\end{array}
$$

In the above calculation, we discovered another clause for the definition of *exec*. In this case, in order to apply the induction hypothesis for y we had to obtain a term of the form

$$
\begin{array}{l}
\mathbf{do}\ n \leftarrow eval\ y \\
\quad exec\ c'\ (n : s')
\end{array}
$$

for some code c' and stack s'. That is, we needed to find suitable instantiations for the variables c' and s' to solve the equation $exec\ c'\ (n : s') \cong_i exec\ c\ ((m+n) : s)$. We achieved this by solving the equality $exec\ c'\ (n : s') = exec\ c\ ((m+n) : s)$ with the instantiations $c' = ADD\ c$ and $s' = m : s$, where $ADD :: Code \rightarrow Code$ is a new constructor for the code datatype.

In the loop case, we use proof rule (2) to apply the induction hypothesis for all j smaller than i:

Case: $e = Loop$

$$
\begin{array}{ll}
& \mathbf{do}\ v \leftarrow eval\ Loop \\
& \quad exec\ c\ (v : s) \\
\cong_i & \{\ \text{definition of } eval\ \} \\
& \mathbf{do}\ v \leftarrow Later\ (eval\ Loop) \\
& \quad exec\ c\ (v : s) \\
\cong_i & \{\ \text{definition of } \gg=\ \} \\
& Later\ (\mathbf{do}\ v \leftarrow eval\ Loop \\
& \qquad\qquad exec\ c\ (v : s)) \\
\cong_i & \{\ \text{proof rule (2), induction hypothesis for } Loop \text{ and } j < i\ \} \\
& Later\ (exec\ (comp\ Loop\ c)\ s) \\
\cong_i & \{\ \text{define: } exec\ (LOOP\ c)\ s = Later\ (exec\ (comp\ Loop\ c)\ s)\ \} \\
& exec\ (LOOP\ c)\ s
\end{array}
$$

In the final step above, we discovered another clause for *exec*. In this step, we aimed to solve the equation $exec\ c'\ s \cong_i Later\ (exec\ (comp\ Loop\ c)\ s)$ for some code c'. We achieved this by strengthening bisimilarity to equality and picking the instantiation $c' = LOOP\ c$, so that the equation becomes a clause for the definition of *exec*. However, this definition is unsatisfactory as it invokes the compiler, whereas we normally expect all compilation to take place at compile-time. To eliminate this issue, we can simplify the definition to $exec\ (LOOP\ c)\ s = Later\ (exec\ (LOOP\ c)\ s)$ by using the equation $comp\ Loop\ c = LOOP\ c$, which we have just derived by means of the above calculation. In addition, we can then further simplify the definitions by eliminating the continuation argument to *LOOP*, as it is not used in the definition of *exec* apart from passing it along to *LOOP* again. That is, we refine the last step of the calculation to the following:

Target language

$$\textbf{data}\ Code = PUSH\ Int\ Code \mid ADD\ Code \mid LOOP \mid HALT$$

Compiler

$$\begin{aligned}
&compile :: Expr \to Code\\
&compile\ e = comp\ e\ HALT\\
&comp :: Expr \to Code \to Code\\
&comp\ (Val\ n)\quad\ \ c = PUSH\ n\ c\\
&comp\ (Add\ x\ y)\ c = comp\ x\ (comp\ y\ (ADD\ c))\\
&comp\ Loop\qquad\ c = LOOP
\end{aligned}$$

Virtual machine

$$\begin{aligned}
&exec :: Code \to Stack \to Partial\ Stack\\
&exec\ (PUSH\ n\ c)\ s &&= exec\ c\ (n : s)\\
&exec\ (ADD\ c)\ (n : m : s) &&= exec\ c\ ((m + n) : s)\\
&exec\ LOOP\ s &&= Later\ (exec\ LOOP\ s)\\
&exec\ HALT\ s &&= return\ s
\end{aligned}$$

Fig. 1. Compiler and virtual machine for the simple expression language.

$$\begin{aligned}
&\quad Later\ (exec\ (comp\ Loop\ c)\ s)\\
\cong_i&\quad \{\ \text{define: } exec\ LOOP\ s = Later\ (exec\ LOOP\ s)\ \}\\
&\quad exec\ LOOP\ s
\end{aligned}$$

All steps of the above compiler calculation can be replicated using the original methodology of Bahr and Hutton [2015], with the exception of the use of proof rule (2). The key novelty is that we are not proceeding by structural induction on the expression e, but rather by induction on both the step index i and the expression e. This allows us to apply the induction hypothesis to an expression that is not structurally smaller, in this case calculating the *Loop* case by using the induction hypothesis for *Loop* itself, with the step-indexing machinery resolving the circularity.

Finally, we conclude by considering the top-level compilation function $compile :: Expr \to Code$, whose correctness can be captured by the following property,

$$\begin{array}{l}\textbf{do}\ v \leftarrow eval\ e\\ \quad\ return\ (v : s)\end{array} \quad\cong\quad exec\ (compile\ e)\ s$$

Using the correctness of *comp*, it is easy to calculate the definition for *compile*:

$$\begin{aligned}
&\quad \textbf{do}\ v \leftarrow eval\ e\\
&\quad\quad return\ (v : s)\\
\cong&\quad \{\ \text{define: } exec\ HALT\ s = return\ s\ \}\\
&\quad \textbf{do}\ v \leftarrow eval\ e\\
&\quad\quad exec\ HALT\ (v : s)\\
\cong&\quad \{\ \text{Theorem 2.1}\ \}\\
&\quad exec\ (comp\ e\ HALT)\ s
\end{aligned}$$

In summary, we have calculated the definitions in Figure 1. Note that *exec* is not total because the case for addition requires a stack of at least two values, but we are free to add equations to the definition to make it total. To do so, we arbitrarily choose to add the catch-all equation $exec\ \_\ \_ = never$ that returns the non-terminating computation *never*, but the choice is not important as it plays no role in the proof of the compiler correctness theorem. In particular, we only ever execute well-formed code produced by the compiler. At first glance, it might also seem that *exec* is not tail-recursive because of the case for *LOOP*. However, *Later* is an effect of the partiality monad that is performed *before* the recursive call. To see this, we can rewrite the right-hand side into the equivalent form $Later\ (return\ ()) \gg exec\ LOOP\ s$. More generally, as we will see in section 4, the semantics of a virtual machine is described by a set of mutually tail-recursive functions.

2.6 Reflection

We conclude this section with some reflective remarks on our new methodology.

Full correctness. The compiler that we have derived for the simple expression language captures both the convergent and divergent aspects of compiler correctness by construction. In particular, compiled code can produce precisely the same result values as the source semantics, no more and no less. Moreover, our methodology only required a single calculation process to establish both aspects of compiler correctness at the same time. To the best of our knowledge, this is the first approach to compiler calculation for non-terminating languages that ensures full correctness.

Extra steps. As noted earlier in this section, the use of strong bisimilarity to formulate compiler correctness in Theorem 2.1 means that we may need to insert extra *Later* steps in the definition of the virtual machine, in order to ensure that the number of steps matches the source semantics. In the above calculation we only had to do this in one place, namely in the definition $exec\ LOOP\ s = Later\ (exec\ LOOP\ s)$. However, the need to do this fell naturally out of the calculation process, via the use of *Later* in the definition $eval\ Loop = Later\ (eval\ Loop)$ that gives a semantics to the looping primitive, and did not require any additional insight or 'eureka step'.

Methodology. Our approach is a natural generalisation of Bahr and Hutton's [2015] methodology to deal with non-terminating languages. The calculations proceed in a similar way, using the desire to apply induction hypotheses as the driving force for the calculation process, from which the compilation machinery then arises in a natural manner. The difference is that we now use bisimilarity rather than equality as the basis for the reasoning, and also exploit the monad laws for the partiality monad. Moreover, whereas previous work required moving to a relational semantics to deal with non-termination, using the partiality monad to make divergence explicit allows us to retain the use of a functional semantics. Crucially, our generalisation to a monadic semantics also maintains a key property of Bahr and Hutton's approach, namely that almost all steps of the calculation are mechanical and do not require creative insight. Only a few steps require some creativity, when we need to solve an equation by introducing a clause for *exec*. It is in these steps that we make design decisions that influence the resulting compiler and virtual machine.

Induction/co-induction. The definitions of *eval* and *exec* combine structural recursion in the *Add* case with productive co-recursion in the *Loop* case. Correspondingly, the compiler calculation combines inductive reasoning for *Add* with co-inductive reasoning for *Loop*. There are different frameworks for achieving this mixing of inductive and co-inductive reasoning methods, such as guarded recursion [Møgelberg and Veltri 2019] and sized types [Hughes et al. 1996]. To simplify the presentation, we have chosen a step-indexed approach, which turns the mixed induction/co-induction proof into a single induction proof and requires no additional formal device.

3 LAMBDA CALCULUS

To demonstrate that our coinductive technique also works for more sophisticated languages, we consider the untyped, call-by-value lambda calculus extended with integers and addition.

3.1 Syntax and Semantics

For the source language syntax, we use de Bruijn indices to represent bound variables:

$$\mathbf{data}\ Expr = Val\ Int \mid Add\ Expr\ Expr \mid Var\ Int \mid Abs\ Expr \mid App\ Expr\ Expr$$

Informally, $Var\ i$ is the variable with de Bruijn index $i \geqslant 0$, while $Abs\ x$ constructs an abstraction over expression x, and $App\ x\ y$ applies the abstraction that results from evaluating expression x to the value of expression y. For example, the lambda term $\lambda x.\lambda y.x+y$ that adds two integers together is represented by the expression $Abs\ (Abs\ (Add\ (Var\ 1)\ (Var\ 0)))$.

Since the language is untyped, it is not normalising. For example, the term $\Omega = (\lambda x.x\, x)\ (\lambda x.x\, x)$ reduces to itself, and hence loops forever. As in the previous section, we will make the non-totality of the semantics explicit using the partiality monad. In particular, the semantics will be coinductively defined as a function $Expr \to Env \to Partial\ Value$ that evaluates a (possibly open) expression to a value in a given environment. Because the result of evaluating an expression may now be a function, the notion of a value includes both integer values and closures:

data *Value* = *Num Int* | *Clo Expr Env*

A closure comprises an expression t and an environment e that provides values for the free variables in the expression. In turn, an environment can be represented as a list of values,

type *Env* = [*Value*]

with the value of the variable with de Bruijn index i given by indexing into the list at this position using a lookup function that diverges if the variable is not found:

```
lookup :: Int → [a] → Partial a
lookup 0 (x : xs) = return x
lookup i (x : xs) = lookup (i − 1) xs
lookup _ _        = never
```

Using these ideas, the semantics for the language can now be defined as follows:

```
eval :: Expr → Env → Partial Value
eval (Val n)   e = return (Num n)
eval (Add x y) e = do Num m ← eval x e
                      Num n ← eval y e
                      return (Num (m + n))
eval (Var i)   e = lookup i e
eval (Abs x)   e = return (Clo x e)
eval (App x y) e = do Clo x′ e′ ← eval x e
                      v ← eval y e
                      Later (eval x′ (v : e′))
```

We conclude with three remarks about this definition. First of all, note that *eval* is structurally recursive except for the final call $eval\ x'\ (v : e')$ in the application case, which recurses on the expression x' that results from evaluating x. Hence, this is the only place in the definition where we need to guard the recursive call with a *Later* to ensure that *eval* is well-defined.

Secondly, the definition for *eval* uses non-exhaustive pattern matching within the **do** blocks. For example, the generator $Num\ m \leftarrow eval\ x\ e$ in the addition case will fail if the result of evaluating x is not a numeric value. This is permitted in Haskell, provided that the underlying monad is an instance of the *MonadFail* class. If pattern matching fails within a **do** block, then the *fail* function of this class is called, which in the case of *Partial* we define as follows:

```
fail :: String → Partial a
fail _ = never
```

This definition means that if pattern matching within the *eval* function fails, such as the result of $eval\ x\ e$ not being of the required form $Num\ m$, then evaluation diverges. The string parameter to *fail* is used for error messages, but does not concern us here.

And finally, note that for simplicity we represented all forms of undefined behaviour in *eval* in the same way using divergence, whether it be due to the source expression not terminating, being type incorrect, or containing an unbound variable. If we wish to have a more fine-grained notion of undefined behaviour, this can be achieved by simply refining the return type of the *eval* function so that it can represent different forms of undefined behaviour. The accompanying Agda code includes an example of this in a compiler calculation for a lambda calculus extended with exceptions, in which return the type of *eval* is refined to *Partial* (*Maybe Value*).

3.2 Compiler Correctness

Our goal now is to calculate a compiler $comp :: Expr \to Code \to Code$ and a stack machine $exec :: Code \to Conf \to Partial\ Conf$, where *Conf* is the type of configurations for the machine. In the previous example, a machine configuration was simply a stack. But because the semantics now requires an environment, the configuration also includes an environment:

$$\textbf{type}\ Conf = (Stack, Env')$$

However, the machine may require a different form of environment compared to the semantics, so we use a new type Env' for this purpose, defined as list of machine values of type $Value'$:

$$\textbf{type}\ Env' = [\,Value'\,]$$

To convert between semantic and machine values, we assume a function $conv :: Value \to Value'$, which can be lifted to environments by simply mapping over the list of values:

$$\begin{aligned} &conv_{\mathsf{E}} :: Env \to Env' \\ &conv_{\mathsf{E}} = map\ conv \end{aligned}$$

Similarly to *comp*, *Code* and *exec*, the definitions for $Value'$ and *conv* are not given in advance, but will be derived during the compiler calculation. Finally, a stack is initially defined as a list of machine values, with the element type being extended as required during the calculation:

$$\begin{aligned} &\textbf{type}\ Stack = [\,Elem\,] \\ &\textbf{data}\ Elem\ = VAL\ Value' \end{aligned}$$

The above assumptions are the same as in [Bahr and Hutton 2015], except that the source semantics is now defined as a function into *Partial Value*, rather than as a big-step operational semantics. These assumptions make precise what *kind* of machine we wish to derive. As a consequence of making non-termination explicit using the partiality monad, we can now formulate compiler correctness in a manner that captures both convergent and divergent behaviour:

Theorem 3.1 (compiler correctness).

$$\begin{array}{lcl} \textbf{do}\ v \leftarrow eval\ t\ e & & \\ \quad exec\ c\ (VAL\ (conv\ v) : s, conv_{\mathsf{E}}\ e) & \cong & exec\ (comp\ t\ c)\ (s, conv_{\mathsf{E}}\ e) \end{array}$$

This property has the same form as our first example, except that the virtual machine now operates on configurations comprising a stack and an environment. As previously, using the fact that $p \cong q$ iff $p \cong_i q$ for all step counts i, compiler correctness can be established by proving the following by induction on the step count i and the lambda term t:

$$\begin{array}{lcl} \textbf{do}\ v \leftarrow eval\ t\ e & & \\ \quad exec\ c\ (VAL\ (conv\ v) : s, conv_{\mathsf{E}}\ e) & \cong_i & exec\ (comp\ t\ c)\ (s, conv_{\mathsf{E}}\ e) \end{array} \tag{3}$$

3.3 Compiler Calculation

For each case of the lambda term t, we seek to transform the left-hand side of property (3) into the form $exec\ c'\ (s, conv_{\mathsf{E}}\ e)$ for some code c', from which we can then deduce $comp\ t\ c = c'$ as the definition for the compiler in this case. As in the previous example, to calculate the compiler we will need to introduce new constructors into the *Code* type, together with their interpretation by *exec*. Moreover, for this example we will also need to add new constructors to the stack element type *Elem* and machine value type *Value′*, and define the conversion function *conv*.

The case for values follows the same pattern as for simple arithmetic expressions, with the minor addition of applying the conversion function *conv*:

Case: $t = Val\ n$

$$
\begin{array}{ll}
 & \mathbf{do}\ v \leftarrow eval\ (Val\ n)\ e \\
 & \quad exec\ c\ (VAL\ (conv\ v) : s, conv_{\mathsf{E}}\ e) \\
\cong_i & \{ \text{definition of } eval \} \\
 & \mathbf{do}\ v \leftarrow return\ (Num\ n) \\
 & \quad exec\ c\ (VAL\ (conv\ v) : s, conv_{\mathsf{E}}\ e) \\
\cong_i & \{ \text{monad laws} \} \\
 & exec\ c\ (VAL\ (conv\ (Num\ n)) : s, conv_{\mathsf{E}}\ e) \\
\cong_i & \{ \text{define: } conv\ (Num\ n) = Num'\ n \} \\
 & exec\ c\ (VAL\ (Num'\ n) : s, conv_{\mathsf{E}}\ e) \\
\cong_i & \{ \text{define: } exec\ (PUSH\ n\ c)\ (s, e) = exec\ c\ (VAL\ (Num'\ n) : s, e) \} \\
 & exec\ (PUSH\ n\ c)\ (s, conv_{\mathsf{E}}\ e)
\end{array}
$$

The case for addition also proceeds in a similar manner to previously, resulting in a new code constructor *ADD* that adds together two numeric values on the stack, as shown below. We also need to take account of the fact that the semantics diverges if addition is applied to non-numeric values, which is achieved by adding a corresponding failure case to the machine:

$$
\begin{array}{ll}
exec\ (ADD\ c)\ (VAL\ (Num'\ n) : VAL\ (Num'\ m) : s, e) & = exec\ c\ (VAL\ (Num'\ (m+n)) : s, e) \\
exec\ (ADD\ c)\ \_ & = never
\end{array}
$$

The case for variables is straightforward, in which we use n rather than i as the de Bruijn index for the variable, as i is already used for the step index within this calculation:

Case: $t = Var\ n$

$$
\begin{array}{ll}
 & \mathbf{do}\ v \leftarrow eval\ (Var\ n)\ e \\
 & \quad exec\ c\ (VAL\ (conv\ v) : s, conv_{\mathsf{E}}\ e) \\
\cong_i & \{ \text{definition of } eval \} \\
 & \mathbf{do}\ v \leftarrow lookup\ n\ e \\
 & \quad exec\ c\ (VAL\ (conv\ v) : s, conv_{\mathsf{E}}\ e) \\
\cong_i & \{ \text{monad laws, lookup lemma} \} \\
 & \mathbf{do}\ v \leftarrow lookup\ n\ (conv_{\mathsf{E}}\ e) \\
 & \quad exec\ c\ (VAL\ v : s, conv_{\mathsf{E}}\ e) \\
\cong_i & \left\{ \begin{array}{l} \text{define: } exec\ (LOOKUP\ n\ c)\ (s, e') = \mathbf{do}\ v \leftarrow lookup\ n\ e \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad exec\ c\ (VAL\ v : s, e') \end{array} \right\} \\
 & exec\ (LOOKUP\ n\ c)\ (s, conv_{\mathsf{E}}\ e)
\end{array}
$$

The lookup lemma used above states that $\mathit{fmap}\ f\ (\mathit{lookup}\ n\ \mathit{xs}) \cong \mathit{lookup}\ n\ (\mathit{map}\ f\ \mathit{xs})$ and thus applies to the term $\mathit{conv}_{\mathsf{E}}\ e$, which is defined as $\mathit{map}\ \mathit{conv}\ e$. Its use allows us to generalise $\mathit{conv}_{\mathsf{E}}\ e$ to e' in the subsequent step where we define *exec* for *LOOKUP*.

The case for application proceeds in the now familiar way, by introducing new constructors to bring the configuration into the form that is required to apply the induction hypotheses. First of all, to apply the induction hypothesis for the expression x' that results from evaluating the first argument expression x, we save and restore a pair comprising code and an environment on the stack by means of a new stack constructor *CLO* and code constructor *RET*:

Case: $t = \mathit{App}\ x\ y$

$$
\begin{array}{ll}
 & \mathbf{do}\ w \leftarrow \mathit{eval}\ (\mathit{App}\ x\ y)\ e \\
 & \quad \mathit{exec}\ c\ (\mathit{VAL}\ (\mathit{conv}\ w) : s, \mathit{conv}_{\mathsf{E}}\ e) \\
\cong_i & \quad \{ \text{definition of } \mathit{eval} \} \\
 & \mathbf{do}\ w \leftarrow \mathbf{do}\ \mathit{Clo}\ x'\ e' \leftarrow \mathit{eval}\ x\ e \\
 & \qquad\qquad\quad v \leftarrow \mathit{eval}\ y\ e \\
 & \qquad\qquad\quad \mathit{Later}\ (\mathit{eval}\ x'\ (v : e')) \\
 & \quad \mathit{exec}\ c\ (\mathit{VAL}\ (\mathit{conv}\ w) : s, \mathit{conv}_{\mathsf{E}}\ e) \\
\cong_i & \quad \{ \text{monad laws} \} \\
 & \mathbf{do}\ \mathit{Clo}\ x'\ e' \leftarrow \mathit{eval}\ x\ e \\
 & \quad v \leftarrow \mathit{eval}\ y\ e \\
 & \quad w \leftarrow \mathit{Later}\ (\mathit{eval}\ x'\ (v : e')) \\
 & \quad \mathit{exec}\ c\ (\mathit{VAL}\ (\mathit{conv}\ w) : s, \mathit{conv}_{\mathsf{E}}\ e) \\
\cong_i & \quad \{ \text{define: } \mathit{exec}\ \mathit{RET}\ (\mathit{VAL}\ v : \mathit{CLO}\ c\ e : s, \_) = \mathit{exec}\ c\ (\mathit{VAL}\ v : s, e) \} \\
 & \mathbf{do}\ \mathit{Clo}\ x'\ e' \leftarrow \mathit{eval}\ x\ e \\
 & \quad v \leftarrow \mathit{eval}\ y\ e \\
 & \quad w \leftarrow \mathit{Later}\ (\mathit{eval}\ x'\ (v : e')) \\
 & \quad \mathit{exec}\ \mathit{RET}\ (\mathit{VAL}\ (\mathit{conv}\ w) : \mathit{CLO}\ c\ (\mathit{conv}_{\mathsf{E}}\ e) : s, \mathit{conv}_{\mathsf{E}}\ (v : e')) \\
\cong_i & \quad \{ \text{definition of } \gg\!= \} \\
 & \mathbf{do}\ \mathit{Clo}\ x'\ e' \leftarrow \mathit{eval}\ x\ e \\
 & \quad v \leftarrow \mathit{eval}\ y\ e \\
 & \quad \mathit{Later}\ (\mathbf{do}\ w \leftarrow \mathit{eval}\ x'\ (v : e') \\
 & \qquad\qquad\quad \mathit{exec}\ \mathit{RET}\ (\mathit{VAL}\ (\mathit{conv}\ w) : \mathit{CLO}\ c\ (\mathit{conv}_{\mathsf{E}}\ e) : s, \mathit{conv}_{\mathsf{E}}\ (v : e'))) \\
\cong_i & \quad \{ \text{proof rule (2), induction hypothesis for } x' \text{ and } j < i \} \\
 & \mathbf{do}\ \mathit{Clo}\ x'\ e' \leftarrow \mathit{eval}\ x\ e \\
 & \quad v \leftarrow \mathit{eval}\ y\ e \\
 & \quad \mathit{Later}\ (\mathit{exec}\ (\mathit{comp}\ x'\ \mathit{RET})\ (\mathit{CLO}\ c\ (\mathit{conv}_{\mathsf{E}}\ e) : s, \mathit{conv}_{\mathsf{E}}\ (v : e')))
\end{array}
$$

The remainder of the calculation is then driven by the desire to apply the induction hypotheses for the argument expressions x and y. In a similar manner to addition, we also extend the *exec* function with an additional failure case to take account of the fact that the semantics diverges if evaluation of the first argument of an application does not result in a closure:

$$
\begin{array}{ll}
\cong_i & \quad \{ \text{define: } \mathit{exec}\ (\mathit{APP}\ c)\ (\mathit{VAL}\ v : \mathit{VAL}\ (\mathit{Clo}'\ c'\ e') : s, e) = \mathit{Later}\ (\mathit{exec}\ c'\ (\mathit{CLO}\ c\ e : s, v : e')) \} \\
 & \mathbf{do}\ \mathit{Clo}\ x'\ e' \leftarrow \mathit{eval}\ x\ e \\
 & \quad v \leftarrow \mathit{eval}\ y\ e \\
 & \quad \mathit{exec}\ (\mathit{APP}\ c)\ (\mathit{VAL}\ (\mathit{conv}\ v) : \mathit{VAL}\ (\mathit{Clo}'\ (\mathit{comp}\ x'\ \mathit{RET})\ (\mathit{conv}_{\mathsf{E}}\ e')) : s, \mathit{conv}_{\mathsf{E}}\ e)
\end{array}
$$

$\cong_i$ $\{$ define: $conv\ (Clo\ x\ e) = Clo'\ (comp\ x\ RET)\ (conv_{\mathsf{E}}\ e)\ \}$

$\quad$ **do** $Clo\ x'\ e' \leftarrow eval\ x\ e$

$\quad\quad v \leftarrow eval\ y\ e$

$\quad\quad exec\ (APP\ c)\ (VAL\ (conv\ v) : VAL\ (conv\ (Clo\ x'\ e')) : s, conv_{\mathsf{E}}\ e)$

$\cong_i$ $\{$ define: $exec\ (APP\ c)\ \_ = never\ \}$

$\quad$ **do** $u \leftarrow eval\ x\ e$

$\quad\quad v \leftarrow eval\ y\ e$

$\quad\quad exec\ (APP\ c)\ (VAL\ (conv\ v) : VAL\ (conv\ u) : s, conv_{\mathsf{E}}\ e)$

$\cong_i$ $\{$ induction hypothesis for $y\ \}$

$\quad$ **do** $u \leftarrow eval\ x\ e$

$\quad\quad exec\ (comp\ y\ (APP\ c))\ (VAL\ (conv\ u) : s, conv_{\mathsf{E}}\ e)$

$\cong_i$ $\{$ induction hypothesis for $x\ \}$

$\quad exec\ (comp\ x\ (comp\ y\ (APP\ c)))\ (s, conv_{\mathsf{E}}\ e)$

Finally, using the new equation for *conv* introduced above, the case for abstraction proceeds by simply adding a new code constructor *ABS* that puts a closure onto the stack:

Case: $t = Abs\ x$

$\quad$ **do** $v \leftarrow eval\ (Abs\ x)\ e$

$\quad\quad exec\ c\ (VAL\ (conv\ v) : s, conv_{\mathsf{E}}\ e)$

$\cong_i$ $\{$ definition of $eval\ \}$

$\quad$ **do** $v \leftarrow return\ (Clo\ x\ e)$

$\quad\quad exec\ c\ (VAL\ (conv\ v) : s, conv_{\mathsf{E}}\ e)$

$\cong_i$ $\{$ monad laws $\}$

$\quad exec\ c\ (VAL\ (conv\ (Clo\ x\ e)) : s, conv_{\mathsf{E}}\ e)$

$\cong_i$ $\{$ definition of $conv\ \}$

$\quad exec\ c\ (VAL\ (Clo'\ (comp\ x\ RET)\ (conv_{\mathsf{E}}\ e)) : s, conv_{\mathsf{E}}\ e)$

$\cong_i$ $\{$ define: $exec\ (ABS\ c'\ c)\ (s, e) = exec\ c\ (VAL\ (Clo'\ c'\ e) : s, e)\ \}$

$\quad exec\ (ABS\ (comp\ x\ RET)\ c)\ (s, conv_{\mathsf{E}}\ e)$

In summary, we have calculated the definitions in Figure 2. As with the previous example, the top-level compilation function *compile* is defined simply by applying *comp* to a nullary code constructor *HALT* that returns the current configuration of the machine. Note that in the final definition of the virtual machine, the failure cases for *ADD* and *APP* that were introduced for *exec* have been generalised to a single catch-all equation $exec\ \_\ \_ = never$, which also covers the possibility that *RET* may fail if the stack is not of the required form. As previously, however, the catch-all equation plays no role in the compiler correctness theorem.

3.4 Reflection

Full correctness. The resulting compiler and virtual machine for the lambda calculus are essentially the same as those calculated by Bahr and Hutton [2015], except that the machine now explicitly deals with the possibility of failure and divergence using the partiality monad. Moreover, the calculation in the original article only considered the convergence aspect of compiler correctness, whereas our new methodology allows us to simultaneously address both convergence and divergence.

Side conditions. Our new lambda calculus calculation is also conceptually simpler, because we no longer need to keep track of side conditions concerning the evaluation of other expressions, as these are now explicit in the term that is being manipulated. For example, during the case for

Target language

$$
\begin{aligned}
\textbf{data}\ \mathit{Code} = {} & \mathit{HALT} \\
\mid {} & \mathit{PUSH}\ \mathit{Int}\ \mathit{Code} \\
\mid {} & \mathit{ADD}\ \mathit{Code} \\
\mid {} & \mathit{LOOKUP}\ \mathit{Int}\ \mathit{Code} \\
\mid {} & \mathit{ABS}\ \mathit{Code}\ \mathit{Code} \\
\mid {} & \mathit{RET} \\
\mid {} & \mathit{APP}\ \mathit{Code}
\end{aligned}
$$

Compiler

$$
\begin{aligned}
& \mathit{compile} :: \mathit{Expr} \to \mathit{Code} \\
& \mathit{compile}\ e = \mathit{comp}\ e\ \mathit{HALT} \\
& \mathit{comp} :: \mathit{Expr} \to \mathit{Code} \to \mathit{Code} \\
& \mathit{comp}\ (\mathit{Val}\ n)\quad\ \ c = \mathit{PUSH}\ n\ c \\
& \mathit{comp}\ (\mathit{Add}\ x\ y)\ c = \mathit{comp}\ x\ (\mathit{comp}\ y\ (\mathit{ADD}\ c)) \\
& \mathit{comp}\ (\mathit{Var}\ i)\quad\ \ c = \mathit{LOOKUP}\ i\ c \\
& \mathit{comp}\ (\mathit{Abs}\ x)\quad\ c = \mathit{ABS}\ (\mathit{comp}\ x\ \mathit{RET})\ c \\
& \mathit{comp}\ (\mathit{App}\ x\ y)\ c = \mathit{comp}\ x\ (\mathit{comp}\ y\ (\mathit{APP}\ c))
\end{aligned}
$$

Virtual machine

$$
\begin{aligned}
& \textbf{type}\ \mathit{Conf}\ = (\mathit{Stack}, \mathit{Env}') \\
& \textbf{type}\ \mathit{Stack} = [\mathit{Elem}] \\
& \textbf{type}\ \mathit{Env}'\ = [\mathit{Value}']
\end{aligned}
$$

$$
\begin{aligned}
& \textbf{data}\ \mathit{Elem}\ \ = \mathit{VAL}\ \mathit{Value}' \mid \mathit{CLO}\ \mathit{Code}\ \mathit{Env}' \\
& \textbf{data}\ \mathit{Value}' = \mathit{Num}'\ \mathit{Int} \mid \mathit{Clo}'\ \mathit{Code}\ \mathit{Env}'
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{exec} :: \mathit{Code} \to \mathit{Conf} \to \mathit{Partial}\ \mathit{Conf} \\
& \mathit{exec}\ \mathit{HALT}\ (s, e) && = \mathit{return}\ (s, e) \\
& \mathit{exec}\ (\mathit{PUSH}\ n\ c)\ (s, e) && = \mathit{exec}\ c\ (\mathit{VAL}\ (\mathit{Num}'\ n) : s, e) \\
& \mathit{exec}\ (\mathit{ADD}\ c)\ (\mathit{VAL}\ (\mathit{Num}'\ n) : \mathit{VAL}\ (\mathit{Num}'\ m) : s, e) && = \mathit{exec}\ c\ (\mathit{VAL}\ (\mathit{Num}'\ (m + n)) : s, e) \\
& \mathit{exec}\ (\mathit{LOOKUP}\ i\ c)\ (s, e) && = \textbf{do}\ v \leftarrow \mathit{lookup}\ i\ e \\
& && \qquad\ \ \mathit{exec}\ c\ (\mathit{VAL}\ v : s, e) \\
& \mathit{exec}\ (\mathit{ABS}\ c'\ c)\ (s, e) && = \mathit{exec}\ c\ (\mathit{VAL}\ (\mathit{Clo}'\ c'\ e) : s, e) \\
& \mathit{exec}\ \mathit{RET}\ (\mathit{VAL}\ v : \mathit{CLO}\ c\ e : s, \_) && = \mathit{exec}\ c\ (\mathit{VAL}\ v : s, e) \\
& \mathit{exec}\ (\mathit{APP}\ c)\ (\mathit{VAL}\ v : \mathit{VAL}\ (\mathit{Clo}'\ c'\ e') : s, e) && = \mathit{Later}\ (\mathit{exec}\ c'\ (\mathit{CLO}\ c\ e : s, v : e')) \\
& \mathit{exec}\ \_\ \_ && = \mathit{never}
\end{aligned}
$$

Conversion functions

$$
\begin{aligned}
& \mathit{conv} :: \mathit{Value} \to \mathit{Value}' \\
& \mathit{conv}\ (\mathit{Num}\ n) = \mathit{Num}'\ n \\
& \mathit{conv}\ (\mathit{Clo}\ x\ e) = \mathit{Clo}'\ (\mathit{comp}\ x\ \mathit{RET})\ (\mathit{conv}_{\mathsf{E}}\ e)
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{conv}_{\mathsf{E}} :: \mathit{Env} \to \mathit{Env}' \\
& \mathit{conv}_{\mathsf{E}} = \mathit{map}\ \mathit{conv}
\end{aligned}
$$

Fig. 2. Compiler and virtual machine for the lambda calculus.

addition, Bahr and Hutton [2015] have side conditions $x \Downarrow_e \mathit{Num}\ m$ and $y \Downarrow_e \mathit{Num}\ n$, which means that m and n appear as free variables in the calculation. In our new methodology, these conditions are explicit in the term being manipulated by means of the generators $\mathit{Num}\ m \leftarrow \mathit{eval}\ x\ e$ and $\mathit{Num}\ n \leftarrow \mathit{eval}\ y\ e$, which simplifies the reasoning process.

Totality. To show that *exec* is well-defined, it suffices to show that all recursive calls to *exec* are on smaller arguments according to a suitable size measure, or are guarded by *Later*. If we define the size of code as the number of constructors that it contains, and the size of a stack as the sum of the sizes of all the code arguments for the *CLO* constructor, then it is straightforward to show that *exec* is well-founded with respect to the sum of the sizes of its code and stack arguments. This well-foundedness argument is formalised in the supplementary material [Bahr and Hutton 2022].

Initial assumptions. When formulating the compiler correctness property, we have made some initial assumptions about the nature of the machine we wish to derive, namely that it is a stack machine with access to a variable environment. The latter is motivated by the fact that the semantics of the source language also requires a variable environment. The machine that we have calculated from these assumptions is similar to the Zinc Abstract Machine (ZAM) [Grégoire and Leroy 2002]. While both machines utilise an environment and a stack, some instructions of the ZAM are combined into a single instruction in our machine. For example, *APP* combines the *GRAB*, *APPLY* and *PUSHRETADDR* instructions of the ZAM. By varying the initial assumptions about the machine, we can influence the resulting compiler and machine. For example, we could have chosen a register machine setup like Bahr and Hutton [2020] instead.

4 NON-DETERMINISM

For our final example, we consider an expression language that supports exceptions and interrupts. Whereas for our previous examples the semantics was expressed using the partiality monad, for this language the appropriate setting is a *non-determinism* monad. The resulting compiler calculation demonstrates that our monadic methodology is not specific to non-total languages, but can also be used to calculate compilers for languages with other forms of effects. Moreover, we also show how the non-determinism and partiality monads can be combined, which allows compilers to be calculated for languages that are both non-total and non-deterministic.

4.1 Syntax and Semantics

For the purposes of this example, we view an *exception* as an unexpected event that arises within an expression itself during its evaluation, such as a division by zero. In turn, an *interrupt* is an unexpected event that arises from the external environment, such as a timeout. These kind of interrupts are also known as asynchronous exceptions, not to be confused with the hardware notion of interrupts, which are more like asynchronous subroutine calls.

The source language we consider is taken from Hutton and Wright [2007], except that we omit the sequencing operator that was required there for a running example:

$$\mathbf{data}\ \mathit{Expr} = \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr} \mid \mathit{Throw} \mid \mathit{Catch}\ \mathit{Expr}\ \mathit{Expr} \mid \mathit{Block}\ \mathit{Expr} \mid \mathit{Unblock}\ \mathit{Expr}$$

While this language does not provide features that are necessary for actual programming, it does provide just enough structure to explore the basic semantics of exceptions and interrupts. In particular, integers and addition provide a minimal language in which to consider normal (non-exceptional) computation, throw and catch constitute a minimal extension in which computations can involve exceptions, and finally, block and unblock will allow us to consider interrupts.

As we shall see, the presence of interrupts in the language means that an expression may evaluate to more than one possible result value. To take account of this, we define a type *ND a* for *non-deterministic* computations that return result values of type *a*:

$$\mathbf{data}\ \mathit{ND}\ a = \emptyset \mid \mathit{Ret}\ a \mid \mathit{ND}\ a \oplus \mathit{ND}\ a$$

The idea is that *Ret* transforms a value into a computation that simply returns this value, while $\oplus$ is a non-deterministic choice operator with $\emptyset$ as its identity element. The non-determinism type forms a monad, with the operations defined as follows:

$$\begin{aligned}
&\mathit{return} :: a \to \mathit{ND}\ a\\
&\mathit{return}\ x = \mathit{Ret}\ x\\
&(\gg\!=) :: \mathit{ND}\ a \to (a \to \mathit{ND}\ b) \to \mathit{ND}\ b\\
&\emptyset \gg\!= f = \emptyset\\
&(\mathit{Ret}\ x) \gg\!= f = f\ x\\
&(p \oplus q) \gg\!= f = (p \gg\!= f) \oplus (q \gg\!= f)
\end{aligned}$$

We'll consider laws for *ND* later in this section. Because the result of evaluating an expression may now be an exception, the notion of a result value is defined using the *Maybe* type:

$$\mathbf{type}\ \mathit{Value} = \mathit{Maybe}\ \mathit{Int}$$

That is, a value is either *Nothing*, which we view as an exceptional value, or has the form *Just n*, which we view as a normal value [Spivey 1990]. The semantics also requires the notion of an interrupt status, which specifies whether interrupts are currently blocked or unblocked:

data *Status* = *B* | *U*

The form of interrupts that we consider is a 'worst-case scenario' in which evaluation of an expression can be interrupted at any point, provided that interrupts are not blocked. In order to realise this behaviour, we define a simple function *interrupt* that has no effect if interrupts are blocked, and otherwise returns the exceptional value *Nothing*:

$$
\begin{array}{l}
\mathit{interrupt} :: \mathit{Status} \to \mathit{ND}\ \mathit{Value} \\
\mathit{interrupt}\ B = \emptyset \\
\mathit{interrupt}\ U = \mathit{return}\ \mathit{Nothing}
\end{array}
$$

To streamline the definition of the semantics, we adopt an extension of the pattern matching syntax for the **do** notation, inspired by a similar syntactic shorthand in Idris [Brady 2013]:

$$
\begin{array}{l}
\mathbf{do}\ p \leftarrow \mathit{foo} \mid \mathit{bar} \\
\quad \mathit{rest}
\end{array}
\qquad \text{which is short for} \qquad
\begin{array}{l}
\mathbf{do}\ x \leftarrow \mathit{foo} \\
\quad \mathbf{case}\ x\ \mathbf{of} \\
\quad\quad p \to \mathbf{do}\ \mathit{rest} \\
\quad\quad \_ \to \mathit{bar}
\end{array}
$$

That is, if matching *foo* against the pattern *p* fails, then evaluation proceeds with *bar* instead of *rest*. Using the above ideas, the semantics can now be defined by mutually recursive functions that evaluate an expression in a given interrupt status to produce a non-deterministic value:

$$
\begin{array}{l}
\mathit{eval} :: \mathit{Expr} \to \mathit{Status} \to \mathit{ND}\ \mathit{Value} \\
\mathit{eval}\ e\ i = \mathit{eval}'\ e\ i \oplus \mathit{interrupt}\ i
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{eval}' :: \mathit{Expr} \to \mathit{Status} \to \mathit{ND}\ \mathit{Value}} \\
\mathit{eval}'\ (\mathit{Val}\ n)\ i & = & \mathit{return}\ (\mathit{Just}\ n) \\
\mathit{eval}'\ (\mathit{Add}\ x\ y)\ i & = & \mathbf{do}\ \mathit{Just}\ m \leftarrow \mathit{eval}\ x\ i \mid \mathit{return}\ \mathit{Nothing} \\
 & & \quad\ \ \mathit{Just}\ n \leftarrow \mathit{eval}\ y\ i \mid \mathit{return}\ \mathit{Nothing} \\
 & & \quad\ \ \mathit{return}\ (\mathit{Just}\ (m + n)) \\
\mathit{eval}'\ \mathit{Throw}\ i & = & \mathit{return}\ \mathit{Nothing} \\
\mathit{eval}'\ (\mathit{Catch}\ x\ y)\ i & = & \mathbf{do}\ \mathit{Just}\ n \leftarrow \mathit{eval}\ x\ i \mid \mathit{eval}\ y\ i \\
 & & \quad\ \ \mathit{return}\ (\mathit{Just}\ n) \\
\mathit{eval}'\ (\mathit{Block}\ x)\ i & = & \mathit{eval}\ x\ B \\
\mathit{eval}'\ (\mathit{Unblock}\ x)\ i & = & \mathit{eval}\ x\ U
\end{array}
$$

That is, *eval* either evaluates the expression using *eval′*, or interrupts the current evaluation if the status permits this. In turn, *eval′* defines the semantics of each language feature. In particular, the use of the extended pattern matching syntax expresses that addition propagates an exception thrown in either argument, while catch behaves as its first argument unless it throws an exception, in which case the exception is handled by behaving as its second argument. The functions *eval* and *eval′* capture the same semantics as Hutton and Wright [2007], but defined in a functional manner using the non-determinism monad, rather than as a big-step operational semantics.

Note that the above semantics uses two monads: *ND* and *Maybe*. However, instead of combining them to a single monad, we keep them separate and treat them in different ways, because they serve different purposes. In particular, *ND* captures the ambient effects that are shared by the source and target languages. Thus we treat *ND* in an *abstract* manner using the **do** notation, and are only interested in which laws it satisfies so that we can reason about non-determinism. In contrast, *Maybe* describes an effect, namely exceptions, that the compiler may decide to 'compile away'. That is, we make no assumption on whether the target machine has a built-in mechanism for handling exceptions. Instead, the compiled code may implement exceptions in a particular way. Using explicit pattern matching will allow us to reason about the *concrete* implementation of exceptions.

4.2 Bisimilarity

ND satisfies the monad laws with respect to equality because it is a free monad. However, it does not satisfy certain laws that we would expect, e.g. that $\emptyset$ is the identity for $\oplus$. To achieve this, we quotient *ND* by an appropriate bisimulation relation, in a similar manner to how we quotiented the partiality monad. To this end, we first define when a non-deterministic computation $p :: ND\ a$ converges to a value $v :: a$ using an inductively defined relation $p \Downarrow v$:

$$\frac{}{Ret\ v \Downarrow v} \qquad \frac{p \Downarrow v}{p \oplus q \Downarrow v} \qquad \frac{q \Downarrow v}{p \oplus q \Downarrow v}$$

That is, $Ret\ v$ converges only to v, while $p \oplus q$ converges to any value that p or q converges to. Note that there is no rule for $\emptyset$ because it never produces a value. We then say that p and q are bisimilar, written as $p \cong q$, if they coincide in terms of their convergence behaviours:

$$p \Downarrow v \quad \text{iff} \quad q \Downarrow v \qquad \text{for all } v$$

In this setting, we don't need to make a distinction between weak and strong bisimilarity as there are no *Later* steps. As expected, the *ND* type satisfies the monad laws up to bisimilarity. In addition, the choice primitives satisfy the laws of a commutative, idempotent monoid:

$$\begin{aligned}
(p \oplus q) \oplus r \quad &\cong \quad p \oplus (q \oplus r) && \text{(associativity)}\\
p \oplus \emptyset \quad &\cong \quad p \quad \cong \quad \emptyset \oplus p && \text{(identity)}\\
p \oplus q \quad &\cong \quad q \oplus p && \text{(commutativity)}\\
p \oplus p \quad &\cong \quad p && \text{(idempotence)}
\end{aligned}$$

We also use three laws that capture how choice interacts with monadic bind:

$$\begin{aligned}
\emptyset \gg\!= f \quad &\cong \quad \emptyset && \text{(left zero)}\\
(p \oplus q) \gg\!= f \quad &\cong \quad (p \gg\!= f) \oplus (q \gg\!= f) && \text{(left distributivity)}\\
(p \gg\!= f) \oplus q \quad &\cong \quad p \gg\!= (\lambda x \to f\ x \oplus q) \qquad \text{if } p \not\cong \emptyset && \text{(interchange)}
\end{aligned}$$

The side condition on the interchange law is required because otherwise in the case when $p \cong \emptyset$ the law simplifies to $q \cong \emptyset$, which is not true in general.

4.3 Compiler Correctness

Our goal now is to calculate a compiler $comp :: Expr \to Code \to Code$ and a stack machine $exec :: Code \to Conf \to ND\ Conf$, where $Conf$ is the type of machine configurations. Because the semantics now requires an interrupt status, this is paired with a stack to form the notion of a configuration, while a stack is initially defined as a list of integer values:

type *Conf* = (*Stack*, *Status*)

type *Stack* = [*Elem*]

data *Elem* = *VAL Int*

To specify what it means for the compiler to be correct, we need to take account of the fact that the evaluation may now fail, i.e. result in an exception. To this end, we follow the approach of Bahr and Hutton [2015] and assume an as-yet undefined function $\mathit{fail} :: \mathit{Stack} \to \mathit{Status} \to \mathit{ND}\ \mathit{Conf}$ that captures the behaviour of the machine in the case when an exception is thrown. Using this function, compiler correctness can now be captured as follows:

Theorem 4.1 (compiler correctness).

$$\begin{array}{l} \mathbf{do}\ \mathit{Just}\ v \leftarrow \mathit{eval}\ e\ i \mid \mathit{fail}\ s\ i \\ \quad \mathit{exec}\ c\ (\mathit{VAL}\ v : s, i) \end{array} \quad \cong \quad \mathit{exec}\ (\mathit{comp}\ e\ c)\ (s, i)$$

The left-hand side states that if evaluation succeeds, then the resulting value is pushed onto the stack prior to executing the remaining code. If evaluation results in an exception, control is transferred to the function *fail* to deal with the exception in some appropriate way.

4.4 Compiler Calculation

We proceed to prove Theorem 4.1 by induction on the expression e, seeking to transform the left-hand side of the property into the form $\mathit{exec}\ c'\ (s, i)$ for some code c', from which we can then define $\mathit{comp}\ e\ c = c'$ in this case. Along the way we will introduce new constructors for *Code* and *Elem*, and new equations for *comp*, *exec* and *fail*. The first few steps are the same for each case:

$$\begin{array}{ll}
& \mathbf{do}\ \mathit{Just}\ v \leftarrow \mathit{eval}\ e\ i \mid \mathit{fail}\ s\ i \\
& \quad \mathit{exec}\ c\ (\mathit{VAL}\ v : s, i) \\
\cong & \quad \{\ \text{definition of } \mathit{eval}\ \} \\
& \mathbf{do}\ \mathit{Just}\ v \leftarrow (\mathit{eval}'\ e\ i \oplus \mathit{interrupt}\ i) \mid \mathit{fail}\ s\ i \\
& \quad \mathit{exec}\ c\ (\mathit{VAL}\ v : s, i) \\
\cong & \quad \{\ \text{left distributivity}\ \} \\
& (\mathbf{do}\ \mathit{Just}\ v \leftarrow \mathit{eval}'\ e\ i \mid \mathit{fail}\ s\ i \\
& \quad \mathit{exec}\ c\ (\mathit{VAL}\ v : s, i)) \oplus \\
& (\mathbf{do}\ \mathit{Just}\ v \leftarrow \mathit{interrupt}\ i \mid \mathit{fail}\ s\ i \\
& \quad \mathit{exec}\ c\ (\mathit{VAL}\ v : s, i))
\end{array}$$

At this point, the second argument to $\oplus$ can be simplified by performing case analysis on the interrupt status. In the case when interrupts are blocked, the second argument simplifies to $\emptyset$ using the left zero law, and when interrupts are unblocked it simplifies to $\mathit{fail}\ s\ i$:

$$\begin{array}{ll}
\cong & \quad \{\ \text{define: } \mathit{inter}\ s\ i = \mathbf{if}\ i \equiv B\ \mathbf{then}\ \emptyset\ \mathbf{else}\ \mathit{fail}\ s\ i\ \} \\
& (\mathbf{do}\ \mathit{Just}\ v \leftarrow \mathit{eval}'\ e\ i \mid \mathit{fail}\ s\ i \\
& \quad \mathit{exec}\ c\ (\mathit{VAL}\ v : s, i)) \oplus \mathit{inter}\ s\ i
\end{array}$$

The final step above introduces a function $\mathit{inter} :: \mathit{Stack} \to \mathit{Status} \to \mathit{ND}\ \mathit{Conf}$ that interrupts the current execution if the interrupt status permits this. We continue the calculation by case analysis on e. Some of these cases will make use of the following two simple lemmas:

Lemma 4.2. $\mathit{eval}\ e\ i \ncong \emptyset \ \ \mathit{and}\ \ \mathit{eval}'\ e\ i \ncong \emptyset$

Proof. By straightforward induction on e. □

Lemma 4.3. $\mathit{fail}\ s\ i \oplus \mathit{inter}\ s\ i \ \cong\ \mathit{fail}\ s\ i$

Proof. By case analysis on i. □

We now continue the compiler calculation. The case for values proceeds in a similar manner to previously, except that we now use the special **do** notation for pattern match failure:

Case: $e = \mathit{Val}\ n$

$$
\begin{aligned}
& (\textbf{do}\ \mathit{Just}\ v \leftarrow \mathit{eval}'\ (\mathit{Val}\ n)\ i \mid \mathit{fail}\ s\ i \\
& \qquad \mathit{exec}\ c\ (\mathit{VAL}\ v : s, i)) \oplus \mathit{inter}\ s\ i \\
\cong\ & \quad \{ \text{definition of } \mathit{eval}' \} \\
& (\textbf{do}\ \mathit{Just}\ v \leftarrow \mathit{return}\ (\mathit{Just}\ n) \mid \mathit{fail}\ s\ i \\
& \qquad \mathit{exec}\ c\ (\mathit{VAL}\ v : s, i)) \oplus \mathit{inter}\ s\ i \\
\cong\ & \quad \{ \text{monad laws} \} \\
& \mathit{exec}\ c\ (\mathit{VAL}\ n : s, i) \oplus \mathit{inter}\ s\ i \\
\cong\ & \quad \{ \text{define: } \mathit{exec}\ (\mathit{PUSH}\ n\ c)\ (s, i) = \mathit{exec}\ c\ (\mathit{VAL}\ n : s, i) \oplus \mathit{inter}\ s\ i \} \\
& \mathit{exec}\ (\mathit{PUSH}\ n\ c)\ (s, i)
\end{aligned}
$$

For addition, there are two key changes from our previous examples. First of all, we make use of the interchange law to push $\oplus$ inside the term being manipulated. And secondly, this case gives our first equation for *fail*, which ensures that intermediate result values are removed from the stack when an exception is thrown, an idea that is usually termed ‘unwinding’ the stack:

Case: $e = \mathit{Add}\ x\ y$

$$
\begin{aligned}
& (\textbf{do}\ \mathit{Just}\ v \leftarrow \mathit{eval}'\ (\mathit{Add}\ x\ y)\ i \mid \mathit{fail}\ s\ i \\
& \qquad \mathit{exec}\ c\ (\mathit{VAL}\ v : s, i)) \oplus \mathit{inter}\ s\ i \\
\cong\ & \quad \{ \text{interchange law; Lemma 4.2; case distribution} \} \\
& \textbf{do}\ \mathit{Just}\ v \leftarrow \mathit{eval}'\ (\mathit{Add}\ x\ y)\ i \mid \mathit{fail}\ s\ i \oplus \mathit{inter}\ s\ i \\
& \qquad \mathit{exec}\ c\ (\mathit{VAL}\ v : s, i) \oplus \mathit{inter}\ s\ i \\
\cong\ & \quad \{ \text{Lemma 4.3} \} \\
& \textbf{do}\ \mathit{Just}\ v \leftarrow \mathit{eval}'\ (\mathit{Add}\ x\ y)\ i \mid \mathit{fail}\ s\ i \\
& \qquad \mathit{exec}\ c\ (\mathit{VAL}\ v : s, i) \oplus \mathit{inter}\ s\ i \\
\cong\ & \quad \{ \text{definition of } \mathit{eval}' \} \\
& \textbf{do}\ \mathit{Just}\ v \leftarrow (\textbf{do}\ \mathit{Just}\ m \leftarrow \mathit{eval}\ x\ i \mid \mathit{return}\ \mathit{Nothing} \\
& \qquad\qquad\qquad\quad \mathit{Just}\ n \leftarrow \mathit{eval}\ y\ i\ \mid \mathit{return}\ \mathit{Nothing} \\
& \qquad\qquad\qquad\quad \mathit{return}\ (\mathit{Just}\ (m + n))) \mid \mathit{fail}\ s\ i \\
& \qquad \mathit{exec}\ c\ (\mathit{VAL}\ v : s, i) \oplus \mathit{inter}\ s\ i \\
\cong\ & \quad \{ \text{monad laws} \} \\
& \textbf{do}\ \mathit{Just}\ m \leftarrow \mathit{eval}\ x\ i \mid \mathit{fail}\ s\ i \\
& \qquad \mathit{Just}\ n \leftarrow \mathit{eval}\ y\ i\ \mid \mathit{fail}\ s\ i \\
& \qquad \mathit{exec}\ c\ (\mathit{VAL}\ (m + n) : s, i) \oplus \mathit{inter}\ s\ i \\
\cong\ & \quad \{ \text{define: } \mathit{exec}\ (\mathit{ADD}\ c)\ (\mathit{VAL}\ n : \mathit{VAL}\ m : s, i) = \mathit{exec}\ c\ (\mathit{VAL}\ (m + n) : s, i) \oplus \mathit{inter}\ s\ i \} \\
& \textbf{do}\ \mathit{Just}\ m \leftarrow \mathit{eval}\ x\ i \mid \mathit{fail}\ s\ i \\
& \qquad \mathit{Just}\ n \leftarrow \mathit{eval}\ y\ i\ \mid \mathit{fail}\ s\ i \\
& \qquad \mathit{exec}\ (\mathit{ADD}\ c)\ (\mathit{VAL}\ n : \mathit{VAL}\ m : s, i) \\
\cong\ & \quad \{ \text{define: } \mathit{fail}\ (\mathit{VAL}\ m : s)\ i = \mathit{fail}\ s\ i \} \\
& \textbf{do}\ \mathit{Just}\ m \leftarrow \mathit{eval}\ x\ i \mid \mathit{fail}\ s\ i \\
& \qquad \mathit{Just}\ n \leftarrow \mathit{eval}\ y\ i\ \mid \mathit{fail}\ (\mathit{VAL}\ m : s)\ i \\
& \qquad \mathit{exec}\ (\mathit{ADD}\ c)\ (\mathit{VAL}\ n : \mathit{VAL}\ m : s, i) \\
\cong\ & \quad \{ \text{induction hypothesis for } y \} \\
& \textbf{do}\ \mathit{Just}\ m \leftarrow \mathit{eval}\ x\ i \mid \mathit{fail}\ s\ i \\
& \qquad \mathit{exec}\ (\mathit{comp}\ y\ (\mathit{ADD}\ c))\ (\mathit{VAL}\ m : s, i)
\end{aligned}
$$

$$
\begin{array}{ll}
\cong & \{\text{induction hypothesis for } x\} \\
& \mathit{exec}\ (\mathit{comp}\ x\ (\mathit{comp}\ y\ (\mathit{ADD}\ c)))\ (s, i)
\end{array}
$$

The case for throw is straightforward, and introduces a new equation for *exec* that transfers control to the auxiliary function *fail* when an exception is thrown:

Case: $e = \mathit{Throw}$

$$
\begin{array}{ll}
& (\mathbf{do}\ \mathit{Just}\ v \leftarrow \mathit{eval}'\ \mathit{Throw}\ i \mid \mathit{fail}\ s\ i \\
& \quad\ \mathit{exec}\ c\ (\mathit{VAL}\ v : s, i)) \oplus \mathit{inter}\ s\ i \\
\cong & \{\text{definition of } \mathit{eval}'\} \\
& (\mathbf{do}\ \mathit{Just}\ v \leftarrow \mathit{return}\ \mathit{Nothing} \mid \mathit{fail}\ s\ i \\
& \quad\ \mathit{exec}\ c\ (\mathit{VAL}\ v : s, i)) \oplus \mathit{inter}\ s\ i \\
\cong & \{\text{monad laws}\} \\
& \mathit{fail}\ s\ i \oplus \mathit{inter}\ s\ i \\
\cong & \{\text{Lemma 4.3}\} \\
& \mathit{fail}\ s\ i \\
\cong & \{\text{define: } \mathit{exec}\ \mathit{THROW}\ (s, i) = \mathit{fail}\ s\ i\} \\
& \mathit{exec}\ \mathit{THROW}\ (s, i)
\end{array}
$$

The case for catch introduces the idea of pushing and popping handler code to the stack, which is usually termed ‘marking’ and ‘unmarking’ the stack. The corresponding *MARK* and *UNMARK* instructions require a new stack constructor *HAN* to store and retrieve exception handling code. This case also gives the second equation for *fail*, which transfers control back to the regular execution function *exec* if handler code is found on top of the stack:

Case: $e = \mathit{Catch}\ x\ y$

$$
\begin{array}{ll}
& (\mathbf{do}\ \mathit{Just}\ v \leftarrow \mathit{eval}'\ (\mathit{Catch}\ x\ y)\ i \mid \mathit{fail}\ s\ i \\
& \quad\ \mathit{exec}\ c\ (\mathit{VAL}\ v : s, i)) \oplus \mathit{inter}\ s\ i \\
\cong & \{\text{definition of } \mathit{eval}'\} \\
& (\mathbf{do}\ \mathit{Just}\ v \leftarrow (\mathbf{do}\ \mathit{Just}\ n \leftarrow \mathit{eval}\ x \mid \mathit{eval}\ y\ i \\
& \qquad\qquad\qquad\qquad \mathit{return}\ (\mathit{Just}\ n)) \mid \mathit{fail}\ s\ i \\
& \quad\ \mathit{exec}\ c\ (\mathit{VAL}\ v : s, i)) \oplus \mathit{inter}\ s\ i \\
\cong & \{\text{monad laws}\} \\
& (\mathbf{do}\ \mathit{Just}\ n \leftarrow \mathit{eval}\ x \mid (\mathbf{do}\ \mathit{Just}\ m \leftarrow \mathit{eval}\ y\ i \mid \mathit{fail}\ s\ i \\
& \qquad\qquad\qquad\qquad\qquad\qquad \mathit{exec}\ c\ (\mathit{VAL}\ m : s, i)) \\
& \quad\ \mathit{exec}\ c\ (\mathit{VAL}\ n : s, i)) \oplus \mathit{inter}\ s\ i \\
\cong & \{\text{induction hypothesis for } y\} \\
& (\mathbf{do}\ \mathit{Just}\ n \leftarrow \mathit{eval}\ x \mid \mathit{exec}\ (\mathit{comp}\ y\ c)\ (s, i) \\
& \quad\ \mathit{exec}\ c\ (\mathit{VAL}\ n : s, i)) \oplus \mathit{inter}\ s\ i \\
\cong & \{\text{define: } \mathit{fail}\ (\mathit{HAN}\ (c : s))\ i = \mathit{exec}\ c\ (s, i)\} \\
& (\mathbf{do}\ \mathit{Just}\ n \leftarrow \mathit{eval}\ x \mid \mathit{fail}\ (\mathit{HAN}\ (\mathit{comp}\ y\ c : s))\ i \\
& \quad\ \mathit{exec}\ c\ (\mathit{VAL}\ n : s, i)) \oplus \mathit{inter}\ s\ i \\
\cong & \{\text{define: } \mathit{exec}\ (\mathit{UNMARK}\ c)\ (\mathit{VAL}\ n : \mathit{HAN}\ \_ : s, i) = \mathit{exec}\ c\ (\mathit{VAL}\ n : s, i)\} \\
& (\mathbf{do}\ \mathit{Just}\ n \leftarrow \mathit{eval}\ x \mid \mathit{fail}\ (\mathit{HAN}\ (\mathit{comp}\ y\ c) : s)\ i \\
& \quad\ \mathit{exec}\ (\mathit{UNMARK}\ c)\ (\mathit{VAL}\ n : \mathit{HAN}\ (\mathit{comp}\ y\ c) : s, i)) \oplus \mathit{inter}\ s\ i \\
\cong & \{\text{induction hypothesis for } x\} \\
& \mathit{exec}\ (\mathit{comp}\ x\ (\mathit{UNMARK}\ c))\ (\mathit{HAN}\ (\mathit{comp}\ y\ c) : s, i) \oplus \mathit{inter}\ s\ i
\end{array}
$$

$\cong$ { define: $exec\ (MARK\ c'\ c)\ (s, i) = exec\ c\ (HAN\ c' : s, i) \oplus inter\ s\ i$ }
$exec\ (MARK\ (comp\ y\ c)\ (comp\ x\ (UNMARK\ c)))\ (s, i)$

Finally, the case for blocking interrupts introduces the idea of saving and restoring the current interrupt status using a new stack constructor *STA*. This case also gives the third equation for *fail*, which ensures that the correct status is maintained when an exception is thrown. The case for unblocking interrupts proceeds in the same way, and we omit the details here.

Case: $e = Block\ x$

$(\mathbf{do}\ Just\ v \leftarrow eval'\ (Block\ x)\ i \mid fail\ s\ i$
$\quad exec\ c\ (VAL\ v : s, i)) \oplus inter\ s\ i$
$\cong$ { definition of $eval'$ }
$(\mathbf{do}\ Just\ v \leftarrow eval\ x\ B \mid fail\ s\ i$
$\quad exec\ c\ (VAL\ v : s, i)) \oplus inter\ s\ i$
$\cong$ { interchange law; Lemma 4.2; case distribution }
$\mathbf{do}\ Just\ v \leftarrow eval\ x\ B \mid (fail\ s\ i \oplus inter\ s\ i)$
$\quad exec\ c\ (VAL\ v : s, i) \oplus inter\ s\ i$
$\cong$ { Lemma 4.3 }
$\mathbf{do}\ Just\ v \leftarrow eval\ x\ B \mid fail\ s\ i$
$\quad exec\ c\ (VAL\ v : s, i) \oplus inter\ s\ i$
$\cong$ { define: $exec\ (RESET\ c)\ (VAL\ v : STA\ i : s, B) = exec\ c\ (VAL\ v : s, i) \oplus inter\ s\ i$ }
$\mathbf{do}\ Just\ v \leftarrow eval\ x\ B \mid fail\ s\ i$
$\quad exec\ (RESET\ c)\ (VAL\ v : STA\ i : s, B)$
$\cong$ { define: $fail\ (STA\ i : s)\ B = fail\ s\ i$ }
$\mathbf{do}\ Just\ v \leftarrow eval\ x\ B \mid fail\ (STA\ i : s)\ B$
$\quad exec\ (RESET\ c)\ (VAL\ v : STA\ i : s, B)$
$\cong$ { induction hypothesis for x }
$exec\ (comp\ x\ (RESET\ c))\ (STA\ i : s, B)$
$\cong$ { define: $exec\ (BLOCK\ c)\ (s, i) = exec\ c\ (STA\ i : s, B)$ }
$exec\ (BLOCK\ (comp\ x\ (RESET\ c)))\ (s, i)$

In summary, we have calculated the definitions in Figure 3, from which we can define the top-level compilation function *compile* in the same manner as previously. Note that the equations that we derived for *exec* and *fail* do not yield total definitions, in the first case because some equations require the stack to be of a certain form, and in the second because there is no equation for the empty stack. In our final definitions we have added catch-all cases that return $\emptyset$, but any choice would be fine because the calculation does not depend on it.

4.5 Reflection

The compiler that we have derived for the interrupts language is essentially the same as in [Hutton and Wright 2007], with two key methodological differences. First of all, we have calculated the compiler rather than verifying it, by means of a principled approach that allowed us to discover the basic techniques for compiling exceptions and interrupts in a systematic manner.

Secondly, our new methodology allows us to simultaneously address both the soundness and completeness aspects of compiler correctness. In particular, our correctness theorem captures that compiled code can produce every result that is permitted by the semantics for the language (completeness), and dually, that compiled code will always produce a result that is permitted by

Target language

$$
\begin{aligned}
\textbf{data}\ \mathit{Code} = {} & \mathit{HALT} \\
\mid\ & \mathit{PUSH\ Int\ Code} \\
\mid\ & \mathit{ADD\ Code} \\
\mid\ & \mathit{THROW} \\
\mid\ & \mathit{MARK\ Code\ Code} \\
\mid\ & \mathit{UNMARK\ Code} \\
\mid\ & \mathit{BLOCK\ Code} \\
\mid\ & \mathit{UNBLOCK\ Code} \\
\mid\ & \mathit{RESET\ Code}
\end{aligned}
$$

Compiler

$$
\begin{aligned}
& \mathit{compile} :: \mathit{Expr} \to \mathit{Code} \\
& \mathit{compile}\ e = \mathit{comp}\ e\ \mathit{HALT} \\[4pt]
& \mathit{comp} :: \mathit{Expr} \to \mathit{Code} \to \mathit{Code} \\
& \mathit{comp}\ (\mathit{Val}\ n) && c = \mathit{PUSH}\ n\ c \\
& \mathit{comp}\ (\mathit{Add}\ x\ y) && c = \mathit{comp}\ x\ (\mathit{comp}\ y\ (\mathit{ADD}\ c)) \\
& \mathit{comp}\ \mathit{Throw} && c = \mathit{THROW} \\
& \mathit{comp}\ (\mathit{Catch}\ x\ y) && c = \mathit{MARK}\ (\mathit{comp}\ y\ c)\ (\mathit{comp}\ x\ (\mathit{UNMARK}\ c)) \\
& \mathit{comp}\ (\mathit{Block}\ x) && c = \mathit{BLOCK}\ (\mathit{comp}\ x\ (\mathit{RESET}\ c)) \\
& \mathit{comp}\ (\mathit{Unblock}\ x) && c = \mathit{UNBLOCK}\ (\mathit{comp}\ x\ (\mathit{RESET}\ c))
\end{aligned}
$$

Virtual machine

$$
\begin{aligned}
& \textbf{type}\ \mathit{Conf}\ = (\mathit{Stack}, \mathit{Status}) && \textbf{data}\ \mathit{Elem}\ = \mathit{VAL\ Int} \mid \mathit{HAN\ Code} \mid \mathit{STA\ Status} \\
& \textbf{type}\ \mathit{Stack} = [\mathit{Elem}] && \textbf{data}\ \mathit{Status} = B \mid U
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{exec} :: \mathit{Code} \to \mathit{Conf} \to \mathit{ND\ Conf} \\
& \mathit{exec}\ \mathit{HALT}\ (s, i) && = \mathit{return}\ (s, i) \\
& \mathit{exec}\ (\mathit{PUSH}\ n\ c)\ (s, i) && = \mathit{exec}\ c\ (\mathit{VAL}\ n : s, i) \oplus \mathit{inter}\ s\ i \\
& \mathit{exec}\ (\mathit{ADD}\ c)\ (\mathit{VAL}\ n : \mathit{VAL}\ m : s, i) && = \mathit{exec}\ c\ (\mathit{VAL}\ (m+n) : s, i) \oplus \mathit{inter}\ s\ i \\
& \mathit{exec}\ \mathit{THROW}\ (s, i) && = \mathit{fail}\ s\ i \\
& \mathit{exec}\ (\mathit{MARK}\ c'\ c)\ (s, i) && = \mathit{exec}\ c\ (\mathit{HAN}\ c' : s, i) \oplus \mathit{inter}\ s\ i \\
& \mathit{exec}\ (\mathit{UNMARK}\ c)\ (\mathit{VAL}\ n : \mathit{HAN}\ \_ : s, i) && = \mathit{exec}\ c\ (\mathit{VAL}\ n : s, i) \\
& \mathit{exec}\ (\mathit{BLOCK}\ c)\ (s, i) && = \mathit{exec}\ c\ (\mathit{STA}\ i : s, B) \\
& \mathit{exec}\ (\mathit{UNBLOCK}\ c)\ (s, i) && = \mathit{exec}\ c\ (\mathit{STA}\ i : s, U) \\
& \mathit{exec}\ (\mathit{RESET}\ c)\ (\mathit{VAL}\ v : \mathit{STA}\ i : s, \_) && = \mathit{exec}\ c\ (\mathit{VAL}\ v : s, i) \oplus \mathit{inter}\ s\ i \\
& \mathit{exec}\ \_\ \_ && = \emptyset
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{fail} :: \mathit{Stack} \to \mathit{Status} \to \mathit{ND\ Conf} && \mathit{inter} :: \mathit{Stack} \to \mathit{Status} \to \mathit{ND\ Conf} \\
& \mathit{fail}\ (\mathit{VAL}\ m : s)\ \ i = \mathit{fail}\ s\ i && \mathit{inter}\ s\ B\ = \emptyset \\
& \mathit{fail}\ (\mathit{HAN}\ c : s)\ \ i = \mathit{exec}\ c\ (s, i) && \mathit{inter}\ s\ U = \mathit{fail}\ s\ U \\
& \mathit{fail}\ (\mathit{STA}\ i : s)\ \ \_ = \mathit{fail}\ s\ i \\
& \mathit{fail}\ \_\ \ \_ = \emptyset
\end{aligned}
$$

Fig. 3. Compiler and virtual machine for the interrupts language.

the semantics (soundness). In our previous compiler verification for this language [2007], separate specifications and verifications were required for each part, whereas we have calculated the compiler using a single, unified reasoning process. Bahr [2015] also calculates a compiler for the interrupts language, but relies on proof automation in Coq to discharge side conditions during the calculation. Discharging these side conditions manually is tedious, which makes Bahr's technique unsuitable for tools such as Agda which lack customisable proof automation.

4.6 Combining Non-determinism and Non-termination

We conclude this section by outlining how the compiler calculation methodology can be extended to support languages that are both non-deterministic and non-total. To this end, we first define a monad $ND_\bot$ that combines the effects of ND and $Partial$:

$$
\begin{aligned}
& \textbf{data}\ ND_\bot\ a && = \emptyset \mid \mathit{Ret}\ a \mid \mathit{ND}\ a \oplus \mathit{ND}\ a \mid \mathit{Delay}\ (ND_\bot^\infty\ a) \\
& \textbf{codata}\ ND_\bot^\infty\ a && = \mathit{Later}\ (ND_\bot\ a)
\end{aligned}
$$

This definition nests the inductive type $ND_\perp$ and the coinductive type $ND_\perp^\infty$, such that a value of type $ND_\perp\ a$ is a possibly infinite tree where each infinite branch must contain infinitely many occurrences of *Later*. In this manner, $ND_\perp$ supports non-termination similarly to *Partial*, while still only allowing finite non-determinism by means of the choice operator $\oplus$.

The convergence relation for *ND* can easily be adapted to $ND_\perp$ by step-indexing:

$$\frac{}{Ret\ v \Downarrow_i v} \qquad \frac{p \Downarrow_i v}{p \oplus q \Downarrow_i v} \qquad \frac{q \Downarrow_i v}{p \oplus q \Downarrow_i v} \qquad \frac{p \Downarrow_i v}{Delay\ (Later\ p) \Downarrow_{i+1} v}$$

However, to equip $ND_\perp$ with a suitable notion of strong bisimilarity, we need to explicitly account for divergence. For example, $p \oplus never \Downarrow_i v$ precisely when $p \Downarrow_i v$, but we should not consider $p \oplus never$ and p bisimilar because the former can always diverge whereas the latter may not, e.g. if $p = Ret\ v$. To capture divergent behaviour, we define a step-indexed divergence relation:

$$\frac{p \Uparrow_i}{p \oplus q \Uparrow_i} \qquad \frac{q \Uparrow_i}{p \oplus q \Uparrow_i} \qquad \frac{p \Uparrow_i}{Delay\ (Later\ p) \Uparrow_{i+1}}$$

In turn, given two computations $p, q :: ND_\perp\ a$ and a natural number i, we say that p and q are i-bisimilar, written as $p \cong_i q$, if the following two conditions hold:

$$p \Downarrow_j v \text{ iff } q \Downarrow_j v \quad \text{for all } v \text{ and } j < i \qquad \text{and} \qquad p \Uparrow_j \text{ iff } q \Uparrow_j \quad \text{for all } j < i$$

This notion of bisimilarity for $ND_\perp$ satisfies all the laws for *ND* given in section 4.2, except for the interchange law. The supplementary material [Bahr and Hutton 2022] includes a compiler calculation for the interrupts language extended with the *Loop* primitive from section 2, which uses the $ND_\perp$ monad to account for the non-deterministic and non-total semantics. While similar to the calculation for the interrupts language in section 4.4, it is also slightly different due to the missing interchange law, which also leads to changes in both the resulting compiler and virtual machine.

5 RELATED WORK

In this section we summarise some of the main developments related to our approach to compiler calculation. A more detailed review of related work is provided in [Bahr and Hutton 2015].

Compiler verification. Research in compiler verification has a long history; see [Dave 2003] for a chronological bibliography and [Patterson and Ahmed 2019] for an overview of more recent developments. A landmark result was CompCert [Leroy 2006b], an optimising C compiler with a machine-checked correctness proof. Correctness for this compiler was formulated as a program refinement: each behaviour of the target code, such as producing a certain output or diverging, must have an equivalent behaviour in the source program. For terminating languages, this refinement property is equivalent to weak bisimilarity, as there is precisely one behaviour. For languages with non-determinism, weak bisimilarity is stronger, as it does not allow the compiler to drop behaviours present in the source program. This is appropriate for the interrupts language in section 4, but in many other cases non-determinism is intended as under-specification, and the compiler is free to choose any behaviour exhibited by the source program. A weaker variant $\gtrsim$ of the bisimulation relation $\cong$ in section 4 that supports such reasoning can be defined simply by replacing 'iff' by 'if' in its definition, and satisfies the same laws, plus $p \oplus q \gtrsim p$ to choose an arbitrary behaviour.

More recently, a number of researchers have generalised CompCert's correctness property to account for the fact that compilers rarely translate whole programs but instead translate individual modules, which are then linked with other modules. Moreover, each module might be compiled by a different compiler, or from a different source language. These generalised correctness properties are enabled by a variety of proof techniques including a combined language that embeds the source, intermediate and target languages [Perconti and Ahmed 2014], devising a suitable logical simulation relation [Beringer et al. 2014; Stewart et al. 2015] or parametric bisimulation relation [Neis et al. 2015], and reformulating correctness in a contextual form [Kang et al. 2016].

Non-termination. Big-step and small-step semantics are ubiquitous in operational semantics. However, in their original forms only the latter can distinguish stuck computations from infinite computations. This limitation has been addressed by using coinductive variants of big-step semantics that can account for non-terminating behaviour [Leroy 2006a; Zúñiga and Bel-Enguix 2020].

The partiality monad was introduced by Capretta [2005], who also demonstrated its use to model possibly infinite computations and a general recursion principle. Danielsson [2012] showed that the partiality monad can be used to define the operational semantics of a programming language for the purpose of proving type soundness, as well as proving the correctness of a simple compiler. For the latter, Danielsson defined the semantics of the source and target language using the same monad, so that the compiler correctness property could be formulated by weak bisimilarity. The use of weak bisimilarity is crucial as the number of *Later* steps performed may be different for source programs and target code. However, while suitable for compiler verification, this approach cannot be used for compiler calculation, as weak bisimilarity does not support the use of a transitive reasoning principle in coinductive calculations, as discussed in section 2.3.

Xia et al. [2019] generalised the partiality monad to interaction trees, which allow additional effects to be considered. A rich theory to reason with interaction trees has been developed in Coq, including a framework for coinductive reasoning over weak bisimilarity [Hur et al. 2020]. Using interaction trees, we can extend our methodology to account for further observable effects, e.g. I/O.

Calculational methodology. The idea of calculating with programs [Backhouse 2003] is a powerful technique for verifying programs, and for deriving programs that are correct by construction. Gibbons and Hinze [2011] showed that the techniques carry over to monadic programs, even if the monads are only specified axiomatically. In this article, we extended this idea from equational reasoning to reasoning over other transitive relations, namely bisimilarity and i-bisimilarity.

Our monadic calculation methodology extends the work of Bahr and Hutton [2015], which is limited to total languages. While they also calculate a compiler for the untyped lambda calculus, the correctness theorem on which the calculation is based only makes guarantees about terminating source programs. More recently, Pickard and Hutton [2021] extended this methodology to a dependently typed-setting to account for typed source languages. Bahr [2015] showed how to calculate compilers for non-deterministic languages, but as noted in section 4.5 this approach is complicated by the need to carefully manage side conditions.

6 CONCLUSION AND FURTHER WORK

In this article we have shown how Bahr and Hutton's [2015] approach to compiler calculation can be extended to account for effects such as non-termination and non-determinism using monadic reasoning. Moreover, the monadic approach allows us to maintain the familiar equational reasoning style for calculations. Interesting topics for further work include: dealing with multiple effects by compiling them away one at a time, leading to multi-stage compilers; combining the technique with a dependently-typed approach to compiler calculation [Pickard and Hutton 2021]; and considering how it may be adapted to register-based machines [Bahr and Hutton 2020].

ACKNOWLEDGMENTS

We thank the reviewers for their useful comments and suggestions. This work was funded by EPSRC grant EP/P00587X/1, *Unified Reasoning About Program Correctness and Efficiency.*

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. Department of Computer Science, University of Aarhus.

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

Patrick Bahr. 2015. Calculating Certified Compilers for Non-Deterministic Languages. In *Proceedings of the Symposium on Mathematics of Program Construction.* https://doi.org/10.1007/978-3-319-19797-5_8

Patrick Bahr and Graham Hutton. 2015. Calculating Correct Compilers. *Journal of Functional Programming* 25 (2015). https://doi.org/10.1017/S0956796815000180

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. Supplementary Material for "Monadic Compiler Calculation". https://doi.org/10.5281/zenodo.6794696

Lennart Beringer, Gordon Stewart, Robert Dockins, and Andrew W. Appel. 2014. Verified Compilation for Shared-Memory C. In *Programming Languages and Systems.* https://doi.org/10.1007/978-3-642-54833-8_7

Edwin Brady. 2013. Idris, a General-Purpose Dependently Typed Programming Language: Design and Implementation. *Journal of Functional Programming* 23, 05 (2013). https://doi.org/10.1017/S095679681300018X

Venanzio Capretta. 2005. General Recursion via Coinductive Types. *Logical Methods in Computer Science* 1, 2 (2005). https://doi.org/10.2168/LMCS-1(2:1)2005

Nils Anders Danielsson. 2012. Operational Semantics Using the Partiality Monad. In *Proceedings of the International Conference on Functional Programming.* https://doi.org/10.1145/2364527.2364546

Maulik A. Dave. 2003. Compiler Verification: A Bibliography. *Software Engineering Notes* 28, 6 (2003). https://doi.org/10.1145/966221.966235

Jeremy Gibbons and Ralf Hinze. 2011. Just Do It: Simple Monadic Equational Reasoning. In *Proceedings of the International Conference on Functional Programming.* https://doi.org/10.1145/2034773.2034777

Benjamin Grégoire and Xavier Leroy. 2002. A Compiled Implementation of Strong Reduction. In *Proceedings of the International Conference on Functional Programming.* https://doi.org/10.1145/581478.581501

John Hughes, Lars Pareto, and Amr Sabry. 1996. Proving the Correctness of Reactive Systems Using Sized Types. In *Proceedings of the Symposium on Principles of Programming Languages.* https://doi.org/10.1145/237721.240882

Chung-kil Hur, Paul He, Yannick Zakowski, and Steve Zdancewic. 2020. An Equational Theory for Weak Bisimulation via Generalized Parameterized Coinduction. In *Proceedings of the International Conference on Certified Programs and Proofs.* https://doi.org/10.1145/3372885.3373813

Graham Hutton and Joel Wright. 2007. What is the Meaning of These Constant Interruptions? *Journal of Functional Programming* 17, 6 (2007). https://doi.org/10.1017/S0956796807006363

Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. 2016. Lightweight Verification of Separate Compilation. In *Proceedings of the Symposium Principles of Programming Languages.* https://doi.org/10.1145/2837614.2837642

Xavier Leroy. 2006a. Coinductive Big-Step Operational Semantics. In *Proceedings of the European Symposium on Programming Languages and Systems.* https://doi.org/10.1007/11693024_5

Xavier Leroy. 2006b. Formal Certification of a Compiler Back-End or: Programming a Compiler with a Proof Assistant. In *Proceedings of the Symposium on Principles of Programming Languages.* https://doi.org/10.1145/1111037.1111042

Erik Meijer. 1992. *Calculating Compilers.* PhD Thesis. Katholieke Universiteit Nijmegen.

Rasmus Ejlers Møgelberg and Niccolò Veltri. 2019. Bisimulation As Path Type for Guarded Recursive Types. *Proceedings of the ACM on Programming Languages* 3, POPL (2019). https://doi.org/10.1145/3290317

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

David Park. 1981. Concurrency and Automata on Infinite Sequences. In *Theoretical Computer Science.* https://doi.org/10.1007/BFb0017309

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

James T. Perconti and Amal Ahmed. 2014. Verifying an Open Compiler Using Multi-Language Semantics. In *Proceedings of the European Symposium on Programming Languages and Systems.* https://doi.org/10.1007/978-3-642-54833-8_8

Mitchell Pickard and Graham Hutton. 2021. Calculating Dependently-Typed Compilers. *Proceedings of the ACM on Programming Languages* 5, ICFP (2021). https://doi.org/10.1145/3473587

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

Peter Sestoft. 1997. Deriving a Lazy Abstract Machine. *Journal of Functional Programming* 7, 03 (1997). https://doi.org/10.1017/S0956796897002712

Mike Spivey. 1990. A Functional Theory of Exceptions. *Science of Computer Programming* 14, 1 (1990). https://doi.org/10.1016/0167-6423(90)90056-J

Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *Proceedings of the Symposium on Principles of Programming Languages*. https://doi.org/10.1145/2676726.2676985

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

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2019. Interaction Trees: Representing Recursive and Impure Programs in Coq. *Proceedings of the ACM on Programming Languages* 4, POPL (2019). https://doi.org/10.1145/3371119

Angel Zúñiga and Gemma Bel-Enguix. 2020. Coinductive Natural Semantics for Compiler Verification in Coq. *Mathematics* 8, 9 (2020). https://doi.org/10.3390/math8091573