JFP **30**, e25, 40 pages, 2020.
doi:10.1017/S0956796820000209

Calculating correct compilers II: Return of the register machines

PATRICK BAHR
IT University of Copenhagen, Denmark
(*e-mail:* paba@itu.dk)

GRAHAM HUTTON
University of Nottingham, UK
(*e-mail:* graham.hutton@nottingham.ac.uk)

Abstract

In 'Calculating Correct Compilers' (Bahr & Hutton, 2015), we developed a new approach to calculating compilers directly from specifications of their correctness. Our approach only required elementary reasoning techniques and has been used to calculate compilers for a wide range of language features and their combination. However, the methodology was focused on stack-based target machines, whereas real compilers often target register-based machines. In this article, we show how our approach can naturally be adapted to calculate compilers for register machines.

1 Introduction

Compilers are central to computing, translating high-level programs written by people into low-level programs that machines can execute. However, compilers are also difficult to design and implement, and even more difficult to formally verify. One approach to addressing these problems is to *calculate compilers* directly from specifications of their correctness, thereby eliminating the gap between writing a compiler and verifying that it is correct, resulting in compilers that are *correct by construction*.

The desire to calculate compilers in this manner has been a key objective of the field of program transformation since its earliest days. However, it has traditionally been viewed as an advanced topic that requires specialist knowledge of techniques such as continuations and defunctionalisation. In recent work (Bahr & Hutton, 2015), we developed a simple but general new approach that avoids the need for such techniques. Our approach builds upon previous work in the field, with a number of key differences.

First of all, our approach is based upon the idea of calculating compilers *directly* from high-level specifications of their correctness, rather than indirectly by applying a series of transformations steps to a semantics for the source language. Secondly, it only requires *simple* equational reasoning techniques and avoids the need for more sophisticated concepts such as continuations and defunctionalisation. Thirdly, it provides a principled means to *discover* new compilation techniques and to consider alternative design choices during the calculation process. And finally, it is readily amenable to mechanical *formalisation*, using a proof assistant as an interactive tool for developing and certifying calculations.

The applicability of our new approach to calculating compilers has been demonstrated for a wide range of source language features and their combination, including arithmetic expressions, exceptions, local state, global state, lambda calculus (with various calling conventions), bounded loops, unbounded loops, non-determinism and interrupts. However, this work had an important limitation: it focused on compilers that target *stack-based* machines, whereas real compilers often target *register-based* machines.

As a first step in this direction, we recently demonstrated (Hutton & Bahr, 2017) how the basic methodology that we developed for stack machines could be adapted to register machines, by showing how McCarthy & Painter's (1967) original compiler *verification* for arithmetic expressions could be reworked as a compiler *calculation*. A central aspect of our calculation was the idea of having the target machine 'clean up after itself' by freeing registers that were no longer being used. This idea significantly simplified the calculations, but suffers from two drawbacks. First of all, real machines do not usually do this. And secondly, having to clean up in this manner is problematic when attempting to scale the methodology up to more sophisticated source languages, by requiring the compiler to keep track of what registers need to be freed and when.

In this article, we show how our compiler calculation methodology for stack machines can be adapted to register machines *without* the need to free unused registers. The resulting methodology retains the simplicity of our original approach and scales in a natural manner to more sophisticated source languages. The key idea of our new approach is a pre-ordering on memory that allows us to 'forget' the contents of registers: informally, we have $m \sqsubseteq m'$ if the memory m can be obtained from the memory m' by freeing some registers. This simple idea avoids the need for the machine to clean up after itself.

As in our previous work, we introduce our new methodology using a simple expression language, before showing how it can be applied to more sophisticated source languages, firstly by adding exceptions, and then lambda expressions. All our programs and calculations are written in Haskell, but we only use the basic 'holy trinity' of functional programming, namely recursive types, recursive functions and inductive proofs. All of our calculations have been mechanically checked using the Coq proof assistant; the proof scripts, together with all Haskell code and a range of additional examples, are available as online supplementary material (Bahr & Hutton, 2020).

2 Machine model

We begin by specifying the form of register machine that our compilers will target. Our basic assumption is that the machine has a *memory* that comprises an infinite sequence of *registers*, each of which has a unique name and is either empty or stores a single integer value. To simplify our calculations, we use abstract types *Mem* and *Reg* for memories and register names, and assume the following primitive operations on these types:

$$
\begin{aligned}
&\mathit{empty} :: \mathit{Mem}\\
&\mathit{set} \quad :: \mathit{Reg} \to \mathit{Int} \to \mathit{Mem} \to \mathit{Mem}\\
&\mathit{get} \quad :: \mathit{Reg} \to \mathit{Mem} \to \mathit{Int}\\
&\mathit{first} \quad :: \mathit{Reg}\\
&\mathit{next} \quad :: \mathit{Reg} \to \mathit{Reg}
\end{aligned}
$$

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

Intuitively, *empty* is the empty memory in which all registers contain no value, while *set* and *get*, respectively, update and fetch the value of a register; we will only apply *get* to non-empty registers, with its behaviour for empty registers left unspecified. In turn, *first* is the name of the first register, and *next* returns the name of the next register; the purpose of the latter is to provide a means to produce fresh register names.

Note that there are no operations to *free* a register (make it empty) or check if a register is free. As we shall see, the fact that the machine cannot check if a register is free plays a key role in our methodology. Nonetheless, we need to consider these concepts during the calculation process, for which purpose we assume two meta-level relations:

$$\begin{aligned} \sqsubseteq \quad &\subseteq \mathit{Mem} \times \mathit{Mem} \\ \mathit{freeFrom} \; &\subseteq \mathit{Reg} \times \mathit{Mem} \end{aligned}$$

Intuitively, $m \sqsubseteq m'$ means the memory m can be obtained from the memory m' by freeing zero or more registers, or equivalently, that m' can be obtained from m by setting zero or more free registers. In turn, $\mathit{freeFrom}\; r\; m$ means all registers from r onwards are free in the memory m. The fact that $\sqsubseteq$ and *freeFrom* are meta-level notions means they can be used by ourselves at 'calculation time', but cannot be used by the machine at 'execution time'.

A simple way to realise the above machine model is to represent a register name as a natural number, the memory as a function from register names to their current value and use a special value to represent a register that is empty. In order to simplify our calculations, however, we only require the following properties of the primitive operations and relations and do not need to be concerned with precisely how they are defined:

Property 1 (empty memory)**.**

$\mathit{freeFrom}\; \mathit{first}\; \mathit{empty}$

Property 2 (setting/getting a register)**.**

$\mathit{get}\; r\; (\mathit{set}\; r\; v\; m) \;=\; v$

Property 3 (setting free register leaves previous registers unchanged)**.**

$\mathit{freeFrom}\; r\; m \;\Rightarrow\; m \sqsubseteq \mathit{set}\; r\; v\; m$

Property 4 (setting free register leaves subsequent registers free)**.**

$\mathit{freeFrom}\; r\; m \;\Rightarrow\; \mathit{freeFrom}\; (\mathit{next}\; r)\; (\mathit{set}\; r\; v\; m)$

Property 5 ($\sqsubseteq$ is a pre-ordering)**.**

$\sqsubseteq$ *is reflexive and transitive*

Property 6 (*set* is monotonic)**.**

$m \sqsubseteq m' \;\Rightarrow\; \mathit{set}\; r\; v\; m \sqsubseteq \mathit{set}\; r\; v\; m'$

Property 7 (memory coherence)**.**

$m \sqsubseteq m' \;\wedge\; \mathit{get}\; r\; m \;\mathit{is\ defined} \;\Rightarrow\; \mathit{get}\; r\; m = \mathit{get}\; r\; m'$

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

Intuitively, property 1 states that every register is free in the empty memory; 2–4 capture the basic behaviour of *set* (the given register is changed, previous registers are unchanged and subsequent registers remain free); 5 ensures that $\sqsubseteq$ can be used for inequational reasoning; and finally, 6 and 7 capture the basic behaviour of $\sqsubseteq$ (it is preserved by setting a register and set registers retain their values in larger memories). These properties are not complete but suffice for our calculations and arose naturally during their development.

2.1 Reflection

We conclude this section with three further remarks about the nature of our machine model. We will return to the first two of these points at the end of the next section.

Infinite registers. Real machines usually have a *finite* set of registers, whereas in our model we follow the common practice (Lattner, 2008) of assuming an infinite set, to ensure that we never run out of registers. The process of mapping from an infinite to a finite set of registers is normally handled by a separate register allocation phase after compilation by means of universal algorithm that solves the problem (Chaitin, 1982), so we do not concern ourselves with this here. For the process of register allocation to be sound, we only require that the machine makes no assumptions about the relative location of registers, which can be achieved by simply prohibiting the use of the *next* operation in the machine.

Free registers. Real registers usually always contain a value, which may be initialised to zero or some non-deterministic value, but for the purposes of our calculations we need to be able to distinguish between registers that are currently free and those that are being used. The fact that this distinction can be only be made at the meta-level ensures that the concept of a register being free (empty) or used (non-empty) is only relevant during the calculation process and plays no role in the machine itself.

Other structures. While we assume that a register machine has access to registers as described in the previous section, we do not restrict the machine to *only* use registers. As with real processors, we allow a register machine to have additional memory structures depending on the kind of programming language features it has to implement. For example, to implement function calls, we may expect to have some form of stack available.

3 Arithmetic expressions

To introduce our methodology, we consider a simple source language *Expr* of arithmetic expressions built up from integer values using an addition operator, whose semantics is given by a function *eval* that evaluates an expression to its integer value:

$$\begin{aligned}
&\textbf{data}\ \mathit{Expr} = \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr}\\
&\mathit{eval} :: \mathit{Expr} \to \mathit{Int}\\
&\mathit{eval}\ (\mathit{Val}\ n) \quad = n\\
&\mathit{eval}\ (\mathit{Add}\ x\ y) = \mathit{eval}\ x + \mathit{eval}\ y
\end{aligned}$$

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

Our goal is to develop a compiler for this language which targets a register machine of the form specified in the previous section. Our development proceeds in three steps:

1. Describe the problem to be solved;
2. Specify what it means for the compiler to be correct;
3. Calculate a compiler that satisfies the specification.

3.1 Problem description

We assume the machine will store the result of evaluating an expression in a special register called the *accumulator*. Rather than using a specific register in the memory for this purpose, we find it convenient to factor out the accumulator as a component of the *configuration* of the machine which comprises the current accumulator value and memory:

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

Using these assumptions, we follow the same approach as in our earlier work for stack machines (Bahr & Hutton, 2015) and seek to define three further components that together complete the definition of a compiler for arithmetic expressions:

- A datatype *Code* that represents code for the machine;
- A function $\mathit{compile} :: \mathit{Expr} \to \mathit{Code}$ that compiles expressions to code;
- A function $\mathit{exec} :: \mathit{Code} \to \mathit{Conf} \to \mathit{Conf}$ that gives a semantics for code.

Intuitively, *Code* will provide a set of primitive machine operations for evaluating expressions, *compile* will translate an expression into a sequence of such operations and *exec* will execute code starting from an initial configuration to give a final configuration.

3.2 Compiler correctness

In order to specify what it means for the compiler to be correct, we first extend the ordering on memory to configurations in a pointwise manner:

$$(a, m) \sqsubseteq (a', m') \quad \Leftrightarrow \quad a = a' \ \wedge \ m \sqsubseteq m'$$

That is, two configurations are related if their accumulator values are equal and their memories are related. Moreover, we require the following property of the *exec* function, which states that the $\sqsubseteq$ ordering on configurations is preserved by executing code:

Property 8 (*exec* is monotonic)**.**

$$s \sqsubseteq s' \ \Rightarrow \ \mathit{exec}\ c\ s \sqsubseteq \mathit{exec}\ c\ s'$$

That is, supplying the execution function with a more defined initial configuration, in the sense that some free registers become set and all other registers are unchanged, will result in a more defined final configuration. Intuitively, this property holds because our machine model does not provide an operation to check if a register is free.

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

Using the above notions, the desired relationship between the source language, compiler and target machine can now be captured by the following correctness property:

Specification 1 (compiler correctness)**.**

$$exec\,(compile\,e)\,(a, empty) \sqsupseteq (eval\,e, empty)$$

That is, compiling an expression and then executing the resulting code starting with an arbitrary accumulator and empty memory will give a final configuration in which the accumulator contains the value of the expression. The use of the $\sqsupseteq$ ordering means that final memory is unconstrained, that is, the compiled code can use any registers for temporary storage. Note that to allow the specification to be read from left to right in this manner, and for consistency with our previous work (Bahr & Hutton, 2015; Hutton & Bahr, 2017), it is expressed using the *reverse* ordering $\sqsupseteq$, defined as usual by $x \sqsupseteq y \Leftrightarrow y \sqsubseteq x$.

The above property captures the correctness of the compiler but is not suitable as a basis for calculating the three undefined components. In particular, our methodology is based on the use of induction and as is often the case, we first need to generalise the property we are considering. We begin by generalising from the empty memory to an arbitrary initial memory m, as the use of a specific memory would preclude the use of induction:

$$exec\,(compile\,e)\,(a, m) \sqsupseteq (eval\,e, m)$$

Secondly, in order that the compiler can use registers for temporary storage, we assume that the initial memory m is free from a given register r onwards and generalise to a compilation function $comp :: Expr \to Reg \to Code$ that takes the first free register r as an additional argument, resulting in the following specification:

$$freeFrom\,r\,m \;\Rightarrow\; exec\,(comp\,e\,r)\,(a, m) \sqsupseteq (eval\,e, m)$$

Finally, as in our earlier work for stack machines (Bahr & Hutton, 2015), we further generalise *comp* to take additional code to be executed after the compiled code. The addition of such a 'code continuation' is a key aspect of our methodology and significantly streamlines the resulting calculations. In conclusion, the correctness of the generalised compiler $comp :: Expr \to Reg \to Code \to Code$ is captured as follows:

Specification 2 (generalised compiler correctness)**.**

$$freeFrom\,r\,m \;\Rightarrow\; exec\,(comp\,e\,r\,c)\;(a, m) \sqsupseteq exec\,c\,(eval\,e, m)$$

That is, if the memory is free from a given register onwards, then compiling an expression and executing the resulting code will give a final configuration in which the accumulator has the same value as executing the additional code starting with the value of the expression in the accumulator. The use of $\sqsupseteq$ means that the compiled code can use any registers from r onwards for temporary storage without having to free them afterwards.

In summary, specifications 1 and 2 express the desired relationships between the undefined components, and our goal now is to calculate implementations that satisfy these. Given that the two specifications have four unknowns (*Code*, *compile*, *exec* and *comp*), this may seem like an impossible task. However, as we shall see, it can readily be achieved using structural induction and simple (in)equational reasoning.

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

3.3 Compiler calculation

To calculate the compiler, we proceed directly from specification 2 by structural induction on the expression e. In each case, we assume *freeFrom r m* and aim to transform the right-hand side $exec\ c\ (eval\ e, m)$ of the inequation into the form $exec\ c'\ (a, m)$ for some code c' using a series of equational ($=$) and inequational ($\sqsubseteq$) reasoning steps, from which we can then conclude that the definition $comp\ e\ r\ c = c'$ satisfies the specification in this case. In order to achieve this, we will find that we need to introduce new constructors into the *Code* type, along with their interpretation by the execution function *exec*.

Case: *e* = *Val n* We assume *freeFrom r m* and calculate as follows:

$$\begin{array}{ll} & exec\ c\ (eval\ (Val\ n), m) \\ = & \quad \{\ \text{applying } eval\ \} \\ & exec\ c\ (n, m) \end{array}$$

Now we seem to be stuck, as at this point no further definitions can be applied. However, we are aiming for a term of the form $exec\ c'\ (a, m)$ for some code c'. Hence, to complete the calculation, we need to solve the following inequation:

$$exec\ \boxed{c'}\ (a, m) \sqsupseteq exec\ c\ (n, m)$$

We highlight variables such as $\boxed{c'}$ in displayed formulae to indicate that they are existentially qualified, whereas all other variables in such formulae are assumed to be universally quantified. That is, the above notation is shorthand for

$$\forall a, m, c, n.\ \exists c'.\ exec\ c'\ (a, m) \sqsupseteq exec\ c\ (n, m)$$

The fact that this is an inequation rather than an equation means that we can also manipulate the memory using the $\sqsupseteq$ ordering. However, there is no need for this here, because the same memory m appears on both sides. Hence, it suffices to solve the following equation, which implies the inequation by reflexivity of the ordering:

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

Note that we cannot simply use this equation as a definition for *exec*, because 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 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,

$$LOAD :: Int \to Code \to Code$$

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

$$exec\ (LOAD\ n\ c)\ (a, m) = exec\ c\ (n, m)$$

That is, the code $LOAD\ n\ c$ is executed by loading the integer n into the accumulator and then executing the code c, which motivates the name for the new code constructor. Using the above ideas, the calculation can now be completed in a straightforward manner:

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

$$\begin{aligned} & exec\ c\ (n, m) \\ = \quad & \{\ \text{definition of } exec\ \} \\ & exec\ (LOAD\ n\ c)\ (a, m) \end{aligned}$$

The final term has the desired form $exec\ c'\ (a, m)$, where $c' = LOAD\ n\ c$, from which we conclude that the following definition satisfies specification 2 in the base case:

$$comp\ (Val\ n)\ r\ c = LOAD\ n\ c$$

That is, an integer value is compiled into a single operation that simply loads the value into the accumulator and then continues with the extra code that is supplied. Note that for this simple case all of the reasoning steps were purely equational, and we did not need to use the register argument r or the assumption that the memory is free from r onwards.

Case: $e = Add\ x\ y$ We assume $freeFrom\ r\ m$ and begin in the same way as above:

$$\begin{aligned} & exec\ c\ (eval\ (Add\ x\ y), m) \\ = \quad & \{\ \text{applying } eval\ \} \\ & exec\ c\ (eval\ x + eval\ y, m) \end{aligned}$$

Once again we seem to be stuck, as no further definitions are applicable. However, as we are performing an inductive calculation, we can use the induction hypotheses for x and y. To use the induction hypothesis for the expression y, which is

$$freeFrom\ r'\ m' \ \Rightarrow\ exec\ (comp\ y\ r'\ c')\ (a', m') \sqsupseteq exec\ c'\ (eval\ y, m')$$

we must satisfy the *freeFrom* precondition for some register r' and memory m' and transform the term that is being manipulated into the form $exec\ c'\ (eval\ y, m')$ for some code c'. That is, we need to satisfy the precondition

$$freeFrom\ \boxed{r'}\ \boxed{m'}$$

and solve the following inequation:

$$exec\ \boxed{c'}\ (eval\ y,\ \boxed{m'}) \sqsupseteq exec\ c\ (eval\ x + eval\ y, m)$$

We are free to instantiate r', m' and c' in order to achieve these goals. First of all, we generalise from the specific values $eval\ x$ and $eval\ y$ in the inequation to give:

$$exec\ \boxed{c'}\ (b,\ \boxed{m'}) \sqsupseteq exec\ c\ (a + b, m)$$

As before, we cannot solve this inequation simply using the corresponding equation as a definition for the function *exec*, in this case because the variables c, a and m would be unbound in the body of the definition. However, we are free to instantiate c' and m' in order to solve the inequation. We consider each unbound variable in turn:

- For the code c, the simplest option is to put it into the code c' as we did in the base case, by adding a new constructor to the *Code* datatype along with its interpretation by the *exec* function. If we attempted to put c into the memory m', this would require storing code in the memory, which is not supported by our memory model.
- For the integer a, the simplest option is to put it into the memory m', by assuming that it is stored in a register. If we attempted to put a into the code c', this would

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

require evaluating the expression x at compile time to produce this value, whereas for a compiler we normally expect evaluation to take place at run time.

- For the memory m, the simplest option is also to put it into the memory m', by assuming that m can be derived from m' in some way. If we attempted to put m into the code c', this would require storing the entire memory in the code, which is not the kind of behaviour we normally expect from a compiler.

How should we satisfy the above requirements for a and m? We begin by assuming that m can be derived from m' by simply freeing some registers, that is, that we have $m \sqsubseteq m'$ and split the inequation to be solved into an inequality that manipulates the memory,

$$exec\ c\ (a + b,\ m')\ \sqsupseteq\ exec\ c\ (a + b, m) \tag{1}$$

and an equality that has the same memory on both sides:

$$exec\ \ c'\ \ (b,\ m')\ =\ exec\ c\ (a + b,\ m') \tag{2}$$

This splitting is valid because $x \sqsupseteq z$ follows from $x = y$ and $y \sqsupseteq z$. Inequation (1) above follows from $m \sqsubseteq m'$ using the monotonicity of *exec* (property 8). In order to satisfy the requirement that a is stored in m', that is, we have $get\ r''\ m' = a$ for some register r'', we can use property 2 (setting/getting a register) and define $m' = set\ r''\ a\ m$. In turn, to discharge the assumption $m \sqsubseteq m'$, we can then use property 3 (setting free register leaves previous registers unchanged), which motivates choosing $r'' = r$ in order to satisfy the precondition $freeFrom\ r''\ m$ of this property. It then remains to solve Equation (2), which by substituting $a = get\ r\ m'$ now has the following form:

$$exec\ \ c'\ \ (b,\ m')\ =\ exec\ c\ (get\ r\ \ m'\ + b,\ \ m')$$

The variable r is now unbound on the right-hand side of the equation but can readily be packaged up along with the variable c in the code argument c' by adding a new constructor to the *Code* datatype that takes these two variables as arguments,

$$ADD :: Reg \to Code \to Code$$

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

$$exec\ (ADD\ r\ c)\ (b, m') = exec\ c\ (get\ r\ m' + b, m')$$

That is, executing the code $ADD\ r\ c$ proceeds by adding the value of register r to the accumulator and then executing the code c, hence the choice of name for the new constructor. We now resume our calculation, using the above ideas to transform the term into the required form to apply the induction hypothesis for y:

$$\begin{array}{ll}
 & exec\ c\ (eval\ x + eval\ y, m) \\
\sqsubseteq & \quad \{\ \text{properties 3 and 8}\ \} \\
 & exec\ c\ (eval\ x + eval\ y, set\ r\ (eval\ x)\ m) \\
= & \quad \{\ \text{property 2}\ \} \\
 & exec\ c\ (get\ r\ (set\ r\ (eval\ x)\ m) + eval\ y, set\ r\ (eval\ x)\ m) \\
= & \quad \{\ \text{definition of } exec\ \} \\
 & exec\ (ADD\ r\ c)\ (eval\ y, set\ r\ (eval\ x)\ m) \\
\sqsubseteq & \quad \{\ \text{induction hypothesis for } y\ \} \\
 & exec\ (comp\ y\ \ r'\ \ (ADD\ r\ c))\ (\ a'\ , set\ r\ (eval\ x)\ m)
\end{array}$$

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

In the final step above, we are free to choose the register r' to satisfy the inductive precondition *freeFrom* r' (*set* r (*eval* x) m). Given our top-level assumption *freeFrom* r m, the simplest choice is to take $r' = next\ r$ and apply property 4 (setting free register leaves subsequent registers free). We are also free to choose the new accumulator value a' at this point. With a view to now applying the induction hypothesis for x, which requires that the accumulator contains *eval* x, we take $a' = eval\ x$, resulting in the following term:

$$exec\ (comp\ y\ (next\ r)\ (ADD\ r\ c))\ (eval\ x, set\ r\ (eval\ x)\ m)$$

To use the induction hypothesis for x at this point, which is

$$freeFrom\ r'\ m' \ \Rightarrow\ exec\ (comp\ x\ r'\ c')\ (a', m') \sqsupseteq exec\ c'\ (eval\ x, m')$$

we must find r' and m' that satisfy *freeFrom* r' m' and solve the inequation

$$exec\ \boxed{c'}\ (eval\ x,\ \boxed{m'}) \sqsupseteq exec\ (comp\ y\ (next\ r)\ (ADD\ r\ c))\ (eval\ x, set\ r\ (eval\ x)\ m)$$

As in the case for y, we first generalise the inequation, in this case by abstracting over the value *eval* x and the code *comp* y (*next* r) (*ADD* r c), to give the following:

$$exec\ \boxed{c'}\ (a,\ \boxed{m'}) \sqsupseteq exec\ c\ (a, set\ r\ a\ m)$$

Once again, we cannot solve this inequation by simply using the corresponding equation

$$exec\ \boxed{c'}\ (a,\ \boxed{m'}) = exec\ c\ (a, set\ r\ a\ m)$$

as a definition for *exec*, because the variables c, r and m would be unbound in the body of the definition. However, we are free to instantiate c' and m'. For m, the simplest option is just to take $m' = m$, that is, to equate the two memories, while as previously we can package r and c up in the code argument c' by adding a new constructor to the *Code* datatype,

$$STORE :: Reg \to Code \to Code$$

and defining a new equation for the function *exec*:

$$exec\ (STORE\ r\ c)\ (a, m) = exec\ c\ (a, set\ r\ a\ m)$$

That is, executing the code *STORE* r c proceeds by storing the accumulator value in register r, hence the choice of name for the new constructor. We then continue the calculation:

$$\begin{array}{ll} & exec\ (comp\ y\ (next\ r)\ (ADD\ r\ c))\ (eval\ x, set\ r\ (eval\ x)\ m) \\ = & \quad \{\ \text{definition of } exec\ \} \\ & exec\ (STORE\ r\ (comp\ y\ (next\ r)\ (ADD\ r\ c)))\ (eval\ x, m) \\ \sqsubseteq & \quad \{\ \text{induction hypothesis for } x\ \} \\ & exec\ (comp\ x\ \boxed{r'}\ (STORE\ r\ (comp\ y\ (next\ r)\ (ADD\ r\ c))))\ (\boxed{a'}, m) \end{array}$$

In the final step above, we are free to choose r' and the new accumulator value a' to satisfy the inductive precondition *freeFrom* r' m. The simplest approach is just to take $r' = r$ and $a' = a$, under which the precondition reduces to our top-level assumption *freeFrom* r m, and the machine configuration has the desired form (a, m). The resulting term

$$exec\ (comp\ x\ r\ (STORE\ r\ (comp\ y\ (next\ r)\ (ADD\ r\ c))))\ (a, m)$$

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

now has the form $exec\ c'\ (a, m)$ for some code c', from which we conclude that the following definition satisfies specification 2 in the inductive case:

$$comp\ (Add\ x\ y)\ r\ c = comp\ x\ r\ (STORE\ r\ (comp\ y\ (next\ r))\ (ADD\ r\ c)))$$

That is, the code for addition first computes the value of expression x and stores the resulting value in the first free register r and then computes the value of expression y and adds the contents of register r to the resulting value. Note that when compiling y the next free register becomes $next\ r$, because r itself is used to store the value of x.

Top-level compiler. We conclude the calculation by considering the top-level compilation function $compile :: Expr \to Code$ whose correctness was captured in specification 1. In a similar manner to specification 2, we aim to rewrite the right-hand side $(eval\ e, empty)$ of the inequation into the form $exec\ c\ (a, empty)$ for some code c, from which we can then conclude that the definition $compile\ e = c$ satisfies the specification.

For this case there is no need to use induction as plain calculation suffices, during which we introduce a code constructor $HALT :: Code$ in order to transform the term that is being manipulated into the required form so that specification 2 can be used.

$$\begin{array}{ll} & (eval\ e, empty) \\ = & \quad \{\ \text{define } exec\ HALT\ (a, m) = (a, m)\ \} \\ & exec\ HALT\ (eval\ e, empty) \\ \sqsubseteq & \quad \{\ \text{specification 2, property 1}\ \} \\ & exec\ (comp\ e\ first\ HALT)\ (a, empty) \end{array}$$

The final term now has the form $exec\ c\ (a, empty)$, where $c = comp\ e\ first\ HALT$, from which we conclude that the following definition satisfies specification 1:

$$compile\ e = comp\ e\ first\ HALT$$

Result. In summary, we have calculated the following definitions:

$$\begin{array}{l} \textbf{data}\ Code = LOAD\ Int\ Code \mid STORE\ Reg\ Code \mid ADD\ Reg\ Code \mid HALT \\ \\ compile :: Expr \to Code \\ compile\ e = comp\ e\ first\ HALT \\ \\ comp :: Expr \to Reg \to Code \to Code \\ comp\ (Val\ n)\quad r\ c = LOAD\ n\ c \\ comp\ (Add\ x\ y)\ r\ c = comp\ x\ r\ (STORE\ r\ (comp\ y\ (next\ r))\ (ADD\ r\ c))) \\ \\ exec :: Code \to Conf \to Conf \\ exec\ (LOAD\ n\ c)\ \ (a, m) = exec\ c\ (n, m) \\ exec\ (STORE\ r\ c)\ (a, m) = exec\ c\ (a, set\ r\ a\ m) \\ exec\ (ADD\ r\ c)\quad (a, m) = exec\ c\ (get\ r\ m + a, m) \\ exec\ HALT\qquad (a, m) = (a, m) \end{array}$$

It is straightforward now to prove that the above definition for *exec* is monotonic, as required by property 8; the proof of this monotonicity uses properties 6 and 7. Note that the *Code* datatype above is essentially just a 'list of operations' as we would expect for this example, but using a recursive type for code rather than a list leaves open the possibility of having non-linear code. We will see an example of this idea in Section 4.

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

For reference, below we give the complete, uninterrupted calculation for the main compilation function *comp*. This version corresponds closely to the manner in which our calculations are mechanically checked using the Coq proof assistant. However, for the purpose of actually *producing* the calculations we prefer the narrative-style above, in which the rationale for each step is carefully justified and explained. In this way, we emphasise the process of discovery that is central to our methodology.

Case: *e* = *Val n*

$$
\begin{aligned}
& exec\ c\ (eval\ (Val\ n), m) \\
= \quad & \{ \text{ applying } eval \ \} \\
& exec\ c\ (n, m) \\
= \quad & \{ \text{ definition of } exec \ \} \\
& exec\ (LOAD\ n\ c)\ (a, m)
\end{aligned}
$$

Case: *e* = *Add x y*

$$
\begin{aligned}
& exec\ c\ (eval\ (Add\ x\ y), m) \\
= \quad & \{ \text{ applying } eval \ \} \\
& exec\ c\ (eval\ x + eval\ y, m) \\
\sqsubseteq \quad & \{ \text{ properties } 3 \ \& \ 8 \ \} \\
& exec\ c\ (eval\ x + eval\ y, set\ r\ (eval\ x)\ m) \\
= \quad & \{ \text{ property } 2 \ \} \\
& exec\ c\ (get\ r\ (set\ r\ (eval\ x)\ m) + eval\ y, set\ r\ (eval\ x)\ m) \\
= \quad & \{ \text{ definition of } exec \ \} \\
& exec\ (ADD\ r\ c)\ (eval\ y, set\ r\ (eval\ x)\ m) \\
\sqsubseteq \quad & \{ \text{ induction hypothesis for } y \ \} \\
& exec\ (comp\ y\ (next\ r)\ (ADD\ r\ c))\ (eval\ x, set\ r\ (eval\ x)\ m) \\
= \quad & \{ \text{ definition of } exec \ \} \\
& exec\ (STORE\ r\ (comp\ y\ (next\ r)\ (ADD\ r\ c)))\ (eval\ x, m) \\
\sqsubseteq \quad & \{ \text{ induction hypothesis for } x \ \} \\
& exec\ (comp\ x\ r\ (STORE\ r\ (comp\ y\ (next\ r)\ (ADD\ r\ c))))\ (a, m)
\end{aligned}
$$

3.4 Example

To illustrate the behaviour of the derived compiler and machine, consider the expression $2 + (3 + 4)$, which is written in our language as $Add\ (Val\ 2)\ (Add\ (Val\ 3)\ (Val\ 4))$. Compiling this expression gives the following code sequence:

$$LOAD\ 2\ (STORE\ 0\ (LOAD\ 3\ (STORE\ 1\ (LOAD\ 4\ (ADD\ 1\ (ADD\ 0\ HALT))))))$$

Executing this code from an initial configuration in which the accumulator contains an arbitrary value a and all other registers are empty (denoted here by $-$) proceeds as follows, where the columns show the contents of the relevant registers after each operation:

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

	Acc	*Reg* 0	*Reg* 1
	a	–	–
LOAD 2	2	–	–
STORE 0	2	2	–
LOAD 3	3	2	–
STORE 1	3	2	3
LOAD 4	4	2	3
ADD 1	7	2	3
ADD 0	9	2	3
HALT	9	2	3

That is, the expression is evaluated by first storing the values 2 and 3 in registers 0 and 1, then loading the value 4 into the accumulator and finally adding the contents of registers 1 and 0 to this in turn, leaving the value 9 in the accumulator, as expected.

The above compiler is essentially the same as that developed in Hutton & Bahr (2017), except that the resulting machine is now simpler by virtue of not having to free registers that are no longer being used; we will return to this issue in further detail at the end of the article. The compiler can also be viewed as a lower-level version of the stack machine compiler developed in Bahr & Hutton (2015), in which the use of relative addressing to a stack is replaced by the use of direct addressing to registers.

3.5 Reflection

We conclude this section with some reflective remarks on our approach to calculating a register-based compiler for simple arithmetic expressions.

Memory. At first sight, one might think that our memory model is just a stack in disguise: values appear to be stored in a contiguous region in memory (the stack), and the machine knows where the next free memory location is (the top of the stack). However, this is not the case. First of all, we make no assumptions about the relative locations of registers generated by *next*. That is, registers *r* and *next r* are not necessary adjacent in memory, because to the machine registers are just names, rather than memory locations. This is important as it allows us to take code generated by our compilers and apply a register allocation algorithm that turns register names into CPU registers and memory locations. And secondly, we only know the next free register at compile time via the register parameter to *comp*, and the fact that this register is indeed free is only maintained at the meta-leval via the predicate *freeFrom*. At run time, the machine itself has no knowledge of free registers.

Strategy. When our calculations got stuck, in each case we made progress by attempting to solve inequations of the following general form:

$$exec\ c'\ (a',\ m')\ \sqsupseteq\ exec\ c\ (a, m) \tag{3}$$

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

Compared to solving an equation, solving an inequation may seem more challenging and is certainly less familiar. Our approach was to replace the inequation by a stronger *equation* of the following form, with a side condition on memories:

$$exec\ c'\ (a',\ m')\ =\ exec\ c\ (a,\ m'')\qquad \text{such that } m''\ \sqsupseteq\ m \tag{4}$$

In particular, it follows immediately from property 8 (monotonicity of *exec*) that any solution of Equation (4) is then also a solution of inequation (3):

$$\begin{aligned}
&exec\ c'\ (a', m')\\
=\quad &\{\ (4)\ \}\\
&exec\ c\ (a, m'')\\
\sqsupseteq\quad &\{\ \text{property 8, given } m'' \sqsupseteq m\ \}\\
&exec\ c\ (a, m)
\end{aligned}$$

In essence, the side condition gives us the additional freedom to manipulate the memory by setting some free registers, because we may choose any m'' that satisfies $m'' \sqsupseteq m$. While the form of Equation (4) makes it easier to solve, formulating compiler correctness itself as an inequation (specification 2) avoids having to keep track of these existentially quantified memories and their side conditions during the calculation. Monotonicity of *exec* is the key to this simplification, which in turn is enabled by the use of the $\sqsupseteq$ ordering.

In our calculations in Section 3.3, we used two ways of solving an equation of the form (4), namely by solving one of the following two stronger equations:

$$exec\ c'\ (a',\ m')\ =\ exec\ c\ (a, m) \tag{5}$$

$$exec\ c'\ (a',\ m')\ =\ exec\ c\ (a,\ m')\qquad \text{such that } m'\ \sqsupseteq m \tag{6}$$

By solving (5) we introduce a machine operation that manipulates the memory (such as *LOAD* and *STORE*), whereas solving (6) introduces a machine operation that assumes certain values to be stored in the memory (such as *ADD*).

4 Exceptions

We now extend the language of arithmetic expressions from the previous section with primitives for throwing and catching an exception:

data *Expr* = *Val Int* | *Add Expr Expr* | *Throw* | *Catch Expr Expr*

Intuitively, *Throw* abandons the current evaluation and throws an exception, while *Catch x y* evaluates the expression x unless it throws an exception, in which case it then proceeds to evaluate the *handler* expression y. To define the semantics for this language as an evaluation function, we utilise the familiar *Maybe* type:

data *Maybe a* = *Nothing* | *Just a*

In particular, we view the value *Nothing* as an exceptional result, and a value of the form *Just x* as a successful result (Spivey, 1990). Using this type, our original evaluation function can be revised to deal with the possibility of exceptions as follows:

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

```
eval :: Expr → Maybe Int
eval (Val n)     = Just n
eval (Add x y)   = case eval x of
                     Just n → case eval y of
                       Just m   → Just (n + m)
                       Nothing → Nothing
                     Nothing → Nothing
eval Throw       = Nothing
eval (Catch x y) = case eval x of
                     Just n   → Just n
                     Nothing → eval y
```

Note that addition propagates an exception thrown in either argument. The *eval* function could also be defined in a monadic manner (Wadler, 1992), but while monads are useful for hiding some of the low-level details of effectful computations, when calculating a compiler such details matter, so we prefer to use the non-monadic definition above.

Our goal now is to calculate a compiler for the extended language, building on our experience with arithmetic expressions. Our development proceeds in three steps:

1. Refine the underlying machine model;
2. Refine the specification of compiler correctness;
3. Calculate a compiler that satisfies the specification.

Moreover, we will also refine the calculation process itself, by starting with a *partial* (incomplete) model for the machine, and a *partial* specification for the compiler. The missing components are then also derived during the calculation process.

4.1 Machine model

We begin with two refinements to our machine model. First of all, rather than assuming that registers in the memory store integer values of type *Int*, we use an alternative representation in which these values are wrapped up in a new datatype called *Val*:

```
data Val = VAL {val :: Int}
```

The selector function $val :: Val \to Int$ extracts the underlying integer from a value. Consequently, the *set* and *get* operations on memory now have the following types:

```
set :: Reg → Val → Mem → Mem
get :: Reg → Mem → Val
```

The reason for the above changes is that during the calculation process we will extend the *Val* type with a new constructor, which will be used to handle exceptions. Secondly, rather than assuming that configurations comprise the current accumulator value and memory, we extend this type with a new, but as yet undefined, component type called *Han*:

```
type Conf = (Acc, Han, Mem)
type Han  = ...
```

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

During the calculation, we will find that we need this new component to handle exceptions. We could also begin with the original register value and machine configuration types and observe during the calculation process that we need to extend their definitions in the above manner. Indeed, this is precisely what happened when we undertook this calculation for the first time. We will return to this point at the end of this example.

4.2 Compiler correctness

For arithmetic expressions, the correctness of the generalised compilation function $comp :: Expr \to Reg \to Code \to Code$ was captured by the following property (specification 2):

$$freeFrom\ r\ m \;\Rightarrow\; exec\ (comp\ e\ r\ c)\ (a, m) \sqsupseteq exec\ c\ (eval\ e, m)$$

In the presence of exceptions, this property needs to be modified to take account of the fact that *eval* now has return type *Maybe Int* rather than *Int*, and that the machine configuration is now a triple. If *eval* succeeds, it is straightforward to modify the specification:

$$freeFrom\ r\ m \;\wedge\; eval\ e = Just\ n \;\Rightarrow\; exec\ (comp\ e\ r\ c)\ (a, h, m) \sqsupseteq exec\ c\ (n, h, m)$$

Given that the type *Han* has not yet been defined, we leave the new value h unchanged above. If *eval* fails, it is not yet clear how *comp* should behave, which we make explicit by introducing a new, but as yet undefined, function *fail* to handle this case:

$$freeFrom\ r\ m \;\wedge\; eval\ e = Nothing \;\Rightarrow\; exec\ (comp\ e\ r\ c)\ (a, h, m) \sqsupseteq fail\ e\ c\ h\ m$$

Just as with the function *comp* itself, during the calculation process we aim to derive a definition for *fail* that satisfies the above property. In summary, we now have the following *partial* specification for the new compiler *comp* in terms of an as-yet-undefined type *Han* and function $fail :: Expr \to Code \to Han \to Mem \to Conf$:

$$\begin{aligned} freeFrom\ r\ m \;\Rightarrow\; exec\ (comp\ e\ r\ c)\ (a, h, m) \sqsupseteq\ & \mathbf{case}\ eval\ e\ \mathbf{of} \\ & \quad Just\ n \to exec\ c\ (n, h, m) \\ & \quad Nothing \to fail\ e\ c\ h\ m \end{aligned}$$

We could now proceed to calculate a definition for *comp* from this property by structural induction on e. However, we would soon get stuck. In particular, note that the variables e, c, h and m each appears *twice* in the **case** expression above. This means that in order to use the induction hypotheses, we need to make sure that the instantiations of each of these variables are aligned, which turns out to be problematic.

As observed in our previous work (Bahr & Hutton, 2015), there is a simple solution to this problem that allows the calculation to proceed: we remove the e and c arguments from the *fail* function, as these turn out to be unnecessary. This results in the following revised specification, in which the function *fail* now has type $Han \to Mem \to Conf$:

Specification 3 (generalised compiler correctness)**.**

$$\begin{aligned} freeFrom\ r\ m \;\Rightarrow\; exec\ (comp\ e\ r\ c)\ \ (a, h, m) \sqsupseteq\ & \mathbf{case}\ eval\ e\ \mathbf{of} \\ & \quad Just\ n \to exec\ c\ (n, h, m) \\ & \quad Nothing \to\ fail\ h\ m \end{aligned}$$

Moreover, because the function *fail* now also forms part of the target machine, as with the function *exec* we also require that *fail* preserves the $\sqsubseteq$ ordering:

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

Property 9 (*fail* is monotonic)**.**
$m \sqsubseteq m' \;\Rightarrow\; fail\ h\ m \sqsubseteq fail\ h\ m'$

The right-hand side of this property presumes that the $\sqsubseteq$ ordering is extended to the new form of configurations in the following, pointwise manner:

$$(a, h, m) \sqsubseteq (a', h', m') \;\Leftrightarrow\; a = a' \wedge h = h' \wedge m \sqsubseteq m'$$

4.3 Compiler calculation

We now calculate a definition for *comp* from specification 3 by structural induction on the expression e. In each case, we aim to transform the right-hand side of the inequation into the form $exec\ c'\ (a, h, m)$ for some code c', from which we can then conclude that the definition $comp\ e\ r\ c = c'$ satisfies the specification 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*. For simplicity, we introduce these new components within the calculations as we proceed. Moreover, for this example, we will also need to add a new constructor to the value type *Val* and define the new handler type *Han* and the execution function *fail*.

Case: *e* = *Val n* This case is straightforward:

case $eval\ (Val\ n)$ **of**
  $Just\ n \;\rightarrow exec\ c\ (n, h, m)$
  $Nothing \rightarrow fail\ h\ m$
$=$ { applying *eval* }
$exec\ c\ (n, h, m)$
$=$ { define: $exec\ (LOAD\ n\ c)\ (a, h, m) = exec\ c\ (n, h, m)$ }
$exec\ (LOAD\ n\ c)\ (a, h, m)$

Case: *e* = *Throw* This case proceeds similarly:

case $eval\ Throw$ **of**
  $Just\ n \;\rightarrow exec\ c\ (n, h, m)$
  $Nothing \rightarrow fail\ h\ m$
$=$ { applying *eval* }
$fail\ h\ m$
$=$ { define: $exec\ THROW\ (a, h, m) = fail\ h\ m$ }
$exec\ THROW\ (a, h, m)$

Case: *e* = *Add x y* This case starts in the same manner as the language without exceptions. In particular, to make use of the induction hypothesis for y, which is

$$freeFrom\ r'\ m' \;\Rightarrow\; exec\ (comp\ y\ r'\ c')\ (a', h', m') \sqsupseteq \begin{array}{l}\textbf{case}\ eval\ y\ \textbf{of}\\ \quad Just\ n' \;\rightarrow\; exec\ c'\ (n', h', m')\\ \quad Nothing \rightarrow\; fail\ h'\ m'\end{array}$$

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

we must satisfy the precondition for some register r' and memory m', and transform the term being manipulated into the following form for some code c' and handler h':

$$\begin{array}{l}\textbf{case}\ \mathit{eval}\ y\ \textbf{of}\\ \quad \mathit{Just}\ n' \ \to\ \mathit{exec}\ c'\ (n', h', m')\\ \quad \mathit{Nothing} \to\ \mathit{fail}\ h'\ m'\end{array}$$

We first focus on the *Just* branch and introduce a code constructor *ADD* to bring the accumulator into the form that we need to apply the induction hypothesis:

$$\begin{array}{ll} & \textbf{case}\ \mathit{eval}\ (\mathit{Add}\ x\ y)\ \textbf{of}\\ & \quad \mathit{Just}\ n \ \to \mathit{exec}\ c\ (n, h, m)\\ & \quad \mathit{Nothing} \to \mathit{fail}\ h\ m\\ = & \quad \{\ \text{applying } \mathit{eval}\ \}\\ & \textbf{case}\ \mathit{eval}\ x\ \textbf{of}\\ & \quad \mathit{Just}\ n \to \textbf{case}\ \mathit{eval}\ y\ \textbf{of}\\ & \qquad \mathit{Just}\ n' \ \to \mathit{exec}\ c\ (n + n', h, m)\\ & \qquad \mathit{Nothing} \to \mathit{fail}\ h\ m\\ & \quad \mathit{Nothing} \to \mathit{fail}\ h\ m\\ \sqsubseteq & \quad \{\ \text{properties 3 \& 8}\ \}\\ & \textbf{case}\ \mathit{eval}\ x\ \textbf{of}\\ & \quad \mathit{Just}\ n \to \textbf{case}\ \mathit{eval}\ y\ \textbf{of}\\ & \qquad \mathit{Just}\ n' \ \to \mathit{exec}\ c\ (n + n', h, \mathit{set}\ r\ (\mathit{VAL}\ n)\ m)\\ & \qquad \mathit{Nothing} \to \mathit{fail}\ h\ m\\ & \quad \mathit{Nothing} \to \mathit{fail}\ h\ m\\ = & \quad \{\ \text{property 2}\ \}\\ & \textbf{case}\ \mathit{eval}\ x\ \textbf{of}\\ & \quad \mathit{Just}\ n \to \textbf{case}\ \mathit{eval}\ y\ \textbf{of}\\ & \qquad \mathit{Just}\ n' \ \to \mathit{exec}\ c\ (\mathit{val}\ (\mathit{get}\ r\ (\mathit{set}\ r\ (\mathit{VAL}\ n)\ m)) + n', h, \mathit{set}\ r\ (\mathit{VAL}\ n)\ m)\\ & \qquad \mathit{Nothing} \to \mathit{fail}\ h\ m\\ & \quad \mathit{Nothing} \to \mathit{fail}\ h\ m\\ = & \quad \{\ \text{define: } \mathit{exec}\ (\mathit{ADD}\ r\ c)\ (a, h, m) = \mathit{exec}\ c\ (\mathit{val}\ (\mathit{get}\ r\ m) + a, h, m)\ \}\\ & \textbf{case}\ \mathit{eval}\ x\ \textbf{of}\\ & \quad \mathit{Just}\ n \to \textbf{case}\ \mathit{eval}\ y\ \textbf{of}\\ & \qquad \mathit{Just}\ n' \ \to \mathit{exec}\ (\mathit{ADD}\ r\ c)\ (n', h, \mathit{set}\ r\ (\mathit{VAL}\ n)\ m)\\ & \qquad \mathit{Nothing} \to \mathit{fail}\ h\ m\\ & \quad \mathit{Nothing} \to \mathit{fail}\ h\ m\end{array}$$

Now the inner *Just* branch matches the form of the induction hypothesis. However, in order to apply the induction hypothesis, the corresponding *Nothing* branch must have the same memory argument, $\mathit{set}\ r\ (\mathit{VAL}\ n)\ m$, rather than just m. This can readily be achieved by using properties 3 and 9, similarly to the second step above. The calculation then concludes in the same manner as the language without exceptions:

$$\begin{array}{l}\textbf{case}\ \mathit{eval}\ x\ \textbf{of}\\ \quad \mathit{Just}\ n \to \textbf{case}\ \mathit{eval}\ y\ \textbf{of}\\ \qquad \mathit{Just}\ n' \ \to \mathit{exec}\ (\mathit{ADD}\ r\ c)\ (n', h, \mathit{set}\ r\ (\mathit{VAL}\ n)\ m)\\ \qquad \mathit{Nothing} \to \mathit{fail}\ h\ m\end{array}$$

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

$$
\begin{array}{ll}
 & \quad \mathit{Nothing} \to \mathit{fail}\ h\ m \\
\sqsubseteq & \quad \{\ \text{properties 3 \& 9}\ \} \\
 & \quad \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
 & \qquad \mathit{Just}\ n \to \textbf{case}\ \mathit{eval}\ y\ \textbf{of} \\
 & \qquad\quad \mathit{Just}\ n' \to \mathit{exec}\ (\mathit{ADD}\ r\ c)\ (n', h, \mathit{set}\ r\ (\mathit{VAL}\ n)\ m) \\
 & \qquad\quad \mathit{Nothing} \to \mathit{fail}\ h\ (\mathit{set}\ r\ (\mathit{VAL}\ n)\ m) \\
 & \qquad \mathit{Nothing} \to \mathit{fail}\ h\ m \\
\sqsubseteq & \quad \{\ \text{induction hypothesis for } y\ \} \\
 & \quad \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
 & \qquad \mathit{Just}\ n \to \mathit{exec}\ (\mathit{comp}\ y\ (\mathit{next}\ r)\ (\mathit{ADD}\ r\ c))\ (n, h, \mathit{set}\ r\ (\mathit{VAL}\ n)\ m) \\
 & \qquad \mathit{Nothing} \to \mathit{fail}\ h\ m \\
= & \quad \{\ \text{define: } \mathit{exec}\ (\mathit{STORE}\ r\ c)\ (a, h, m) = \mathit{exec}\ c\ (a, h, \mathit{set}\ r\ (\mathit{VAL}\ a)\ m)\ \} \\
 & \quad \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
 & \qquad \mathit{Just}\ n \to \mathit{exec}\ (\mathit{STORE}\ r\ (\mathit{comp}\ y\ (\mathit{next}\ r)\ (\mathit{ADD}\ r\ c)))\ (n, h, m) \\
 & \qquad \mathit{Nothing} \to \mathit{fail}\ h\ m \\
\sqsubseteq & \quad \{\ \text{induction hypothesis for } x\ \} \\
 & \quad \mathit{exec}\ (\mathit{comp}\ x\ r\ (\mathit{STORE}\ r\ (\mathit{comp}\ y\ (\mathit{next}\ r)\ (\mathit{ADD}\ r\ c))))\ (a, h, m)
\end{array}
$$

Case: *e* = *Catch x y* In this final case of the compiler calculation, applying the evaluation function allows us to apply the induction hypothesis for y:

$$
\begin{array}{ll}
 & \quad \textbf{case}\ \mathit{eval}\ (\mathit{Catch}\ x\ y)\ \textbf{of} \\
 & \qquad \mathit{Just}\ n \to \mathit{exec}\ c\ (n, h, m) \\
 & \qquad \mathit{Nothing} \to \mathit{fail}\ h\ m \\
= & \quad \{\ \text{applying } \mathit{eval}\ \} \\
 & \quad \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
 & \qquad \mathit{Just}\ n \to \mathit{exec}\ c\ (n, h, m) \\
 & \qquad \mathit{Nothing} \to \textbf{case}\ \mathit{eval}\ y\ \textbf{of} \\
 & \qquad\quad \mathit{Just}\ n' \to \mathit{exec}\ c\ (n', h, m) \\
 & \qquad\quad \mathit{Nothing} \to \mathit{fail}\ h\ m \\
\sqsubseteq & \quad \{\ \text{induction hypothesis for } y\ \} \\
 & \quad \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
 & \qquad \mathit{Just}\ n \to \mathit{exec}\ c\ (n, h, m) \\
 & \qquad \mathit{Nothing} \to \mathit{exec}\ (\mathit{comp}\ y\ r\ c)\ (\ a'\ , h, m)
\end{array}
$$

In the final step above, we are free to choose a register r' to satisfy the inductive precondition *freeFrom* $r'\ m$. Given our top-level assumption *freeFrom* $r\ m$, the simplest choice is to take $r' = r$. We are also free to choose the accumulator a' and will return to this shortly.

We now face a similar issue to the *Add* calculation, in that the *Nothing* case does not match the form of the induction hypothesis for x. To match, it must have the form $\mathit{fail}\ h'\ m'$ for some h' and m'. That is, we need to solve the following inequation:

$$\mathit{fail}\ \ h'\ \ m'\ \ \sqsupseteq\ \mathit{exec}\ (\mathit{comp}\ y\ r\ c)\ (\ a'\ , h, m)$$

As with simple arithmetic expressions, we can solve such an inequation by solving the following stronger equation, with a side condition on memories (Section 3.5):

$$\mathit{fail}\ \ h'\ \ m'\ \ =\ \mathit{exec}\ (\mathit{comp}\ y\ r\ c)\ (\ a'\ , h,\ m''\ ) \qquad \text{such that}\ \ m''\ \sqsupseteq m$$

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

There are two reasons why we cannot simply take this equation as a definition for *fail*. First of all, we would not expect the function *fail*, which forms part of the target machine, to invoke the compilation function *comp*. While we could imagine a machine that compiles expressions at run time, we exclude such behaviour here. Therefore, we strengthen the equation by generalising from the specific code *comp y r c* to give:

$$fail\ h'\ m' = exec\ c'\ (a', h, m'') \qquad \text{such that } m'' \sqsupseteq m$$

Secondly, if we now took this equation as a definition for *fail*, the variables c', a', h and m'' would be unbound in the body. However, we are free to instantiate h', m' and a' to solve the equation. Note that since a' occurs on the right-hand side and is unconstrained, we are free to choose an arbitrary integer for it, say 0. We now consider each unbound variable:

- For the memory m'', the simplest option is just to equate it with m', that is, take $m'' = m'$, as a result of which the memory side condition then becomes $m' \sqsupseteq m$.
- For the code c', we have the choice of putting it into either of the h' or m' arguments. Here we choose to put the code into h'. Putting the code into the other argument is possible too and results in a slightly different form of machine.
- For the handler h, the natural choice might seem to be to put it into the h' argument. However, this would mean that *Han*, the type of h and h', then becomes a recursive type in the form of a stack-like structure. For simplicity, we choose to store h in the memory argument m', which avoids the need to introduce a stack type.

How should we satisfy the above requirements for c' and h? First of all, in order to be able to store the handler h in the memory, we add a new constructor

$$HAN :: Han \to Val$$

to the register value type *Val* that takes a handler as an argument, together with a selector function $han :: Val \to Han$ to extract the underlying handler from a value. Assuming that h is stored in the memory m' then means we have $han\ (get\ r'\ m') = h$ for some register r', a free variable which then itself also needs to be stored somewhere. The only sensible choice to store the register r' is in the handler h', as attempting to store it in the memory m' would lead to the circular problem of then needing to store another register r''.

In conclusion, we store both the code c' and register r' in the handler argument h', which motivates taking $h' = (c', r')$ and defining the corresponding type *Han* by:

$$\textbf{type}\ Han = (Code, Reg)$$

Moreover, to ensure $han\ (get\ r'\ m') = h$, we take $m' = set\ r'\ (HAN\ h)\ m$ and appeal to property 2 (setting/getting a register). In turn, to discharge the side condition $m' \sqsupseteq m$, we can use property 3 (setting free register leaves previous registers unchanged), which motivates choosing $r' = r$ to satisfy the *freeFrom* precondition. Putting everything above together, we have derived the following definition for $fail :: Han \to Mem \to Conf$:

$$fail\ (c, r)\ m = exec\ c\ (0, han\ (get\ r\ m), m)$$

That is, when an exception is thrown, we execute the handler code c in a machine configuration in which the accumulator is set to zero, a new handler is fetched from register r and the memory remains as m. Using these ideas, we now resume the calculation:

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

$$
\begin{array}{ll}
 & \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
 & \quad \mathit{Just}\ n \quad \to \mathit{exec}\ c\ (n, h, m) \\
 & \quad \mathit{Nothing} \to \mathit{exec}\ (\mathit{comp}\ y\ r\ c)\ (0, h, m) \\
\sqsubseteq & \quad \{\ \text{properties 3 \& 8}\ \} \\
 & \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
 & \quad \mathit{Just}\ n \quad \to \mathit{exec}\ c\ (n, h, m) \\
 & \quad \mathit{Nothing} \to \mathit{exec}\ (\mathit{comp}\ y\ r\ c)\ (0, h, \mathit{set}\ r\ (\mathit{HAN}\ h)\ m) \\
= & \quad \{\ \text{property 2}\ \} \\
 & \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
 & \quad \mathit{Just}\ n \quad \to \mathit{exec}\ c\ (n, h, m) \\
 & \quad \mathit{Nothing} \to \mathit{exec}\ (\mathit{comp}\ y\ r\ c)\ (0, \mathit{han}\ (\mathit{get}\ r\ (\mathit{set}\ r\ (\mathit{HAN}\ h)\ m)), \mathit{set}\ r\ (\mathit{HAN}\ h)\ m) \\
= & \quad \{\ \text{definition of } \mathit{fail}\ \} \\
 & \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
 & \quad \mathit{Just}\ n \quad \to \mathit{exec}\ c\ (n, h, m) \\
 & \quad \mathit{Nothing} \to \mathit{fail}\ (\mathit{comp}\ y\ r\ c, r)\ (\mathit{set}\ r\ (\mathit{HAN}\ h)\ m)
\end{array}
$$

We have now successfully transformed the *Nothing* case into the required form $\mathit{fail}\ h'\ m'$ to apply the induction hypothesis for x. However, in order to apply this, we also need to align the handler and memory arguments in the two cases, which are currently different. A simple transformation of the *Just* case aligns the two memories:

$$
\begin{array}{ll}
 & \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
 & \quad \mathit{Just}\ n \quad \to \mathit{exec}\ c\ (n, h, m) \\
 & \quad \mathit{Nothing} \to \mathit{fail}\ (\mathit{comp}\ y\ r\ c, r)\ (\mathit{set}\ r\ (\mathit{HAN}\ h)\ m) \\
\sqsubseteq & \quad \{\ \text{properties 3 \& 8}\ \} \\
 & \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
 & \quad \mathit{Just}\ n \quad \to \mathit{exec}\ c\ (n, h, \mathit{set}\ r\ (\mathit{HAN}\ h)\ m) \\
 & \quad \mathit{Nothing} \to \mathit{fail}\ (\mathit{comp}\ y\ r\ c, r)\ (\mathit{set}\ r\ (\mathit{HAN}\ h)\ m)
\end{array}
$$

In turn, a further transformation of the *Just* case by introducing a new code constructor *UNMARK* brings the two handlers into alignment:

$$
\begin{array}{ll}
 & \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
 & \quad \mathit{Just}\ n \quad \to \mathit{exec}\ c\ (n, h, \mathit{set}\ r\ (\mathit{HAN}\ h)\ m) \\
 & \quad \mathit{Nothing} \to \mathit{fail}\ (\mathit{comp}\ y\ r\ c, r)\ (\mathit{set}\ r\ (\mathit{HAN}\ h)\ m) \\
= & \quad \{\ \text{property 2}\ \} \\
 & \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
 & \quad \mathit{Just}\ n \quad \to \mathit{exec}\ c\ (n, \mathit{han}\ (\mathit{get}\ r\ (\mathit{set}\ r\ (\mathit{HAN}\ h)\ m)), \mathit{set}\ r\ (\mathit{HAN}\ h)\ m) \\
 & \quad \mathit{Nothing} \to \mathit{fail}\ (\mathit{comp}\ y\ r\ c, r)\ (\mathit{set}\ r\ (\mathit{HAN}\ h)\ m) \\
= & \quad \{\ \text{define: } \mathit{exec}\ (\mathit{UNMARK}\ c)\ (a, (\_, r), m) = \mathit{exec}\ c\ (a, \mathit{han}\ (\mathit{get}\ r\ m), m)\ \} \\
 & \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
 & \quad \mathit{Just}\ n \quad \to \mathit{exec}\ (\mathit{UNMARK}\ c)\ (n, (\mathit{comp}\ y\ r\ c, r), \mathit{set}\ r\ (\mathit{HAN}\ h)\ m) \\
 & \quad \mathit{Nothing} \to \mathit{fail}\ (\mathit{comp}\ y\ r\ c, r)\ (\mathit{set}\ r\ (\mathit{HAN}\ h)\ m)
\end{array}
$$

The new equation for *exec* above states that executing the code $\mathit{UNMARK}\ c$ proceeds by discarding the current handler, and then executing the code c with the handler replaced by the version stored in the handler register. This process is known as 'unmarking' the handler stack, where in our register-based setting the handlers are stored in registers rather than in a separate stack structure. Finally, we can now apply the induction hypothesis:

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

$$
\begin{array}{ll}
& \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
& \quad \mathit{Just}\ n \quad \to \mathit{exec}\ (\mathit{UNMARK}\ c)\ (n, (\mathit{comp}\ y\ r\ c, r), \mathit{set}\ r\ (\mathit{HAN}\ h)\ m) \\
& \quad \mathit{Nothing} \to \mathit{fail}\ (\mathit{comp}\ y\ r\ c, r)\ (\mathit{set}\ r\ (\mathit{HAN}\ h)\ m) \\
\sqsubseteq & \quad \{\ \text{induction hypothesis for } x\ \&\ \text{property 4}\ \} \\
& \mathit{exec}\ (\mathit{comp}\ x\ (\mathit{next}\ r)\ (\mathit{UNMARK}\ c))\ (a, (\mathit{comp}\ y\ r\ c, r), \mathit{set}\ r\ (\mathit{HAN}\ h)\ m)
\end{array}
$$

In this step, we are free to choose a register r' in order to satisfy the inductive precondition $\mathit{freeFrom}\ r'\ (\mathit{set}\ r\ (\mathit{HAN}\ h)\ m)$. The simplest choice is to take $r' = \mathit{next}\ r$ and apply property 4 (setting free register leaves subsequent registers free). We are also free to choose the new accumulator value a' at this point. With the goal in mind of arriving at a term of the form $\mathit{exec}\ c'\ (a, h, m)$ for some code c', the simplest choice is to take $a' = a$. It is now straightforward to complete the calculation by introducing a new code constructor *MARK* to bring the machine configuration into the required form:

$$
\begin{array}{ll}
& \mathit{exec}\ (\mathit{comp}\ x\ (\mathit{next}\ r)\ (\mathit{UNMARK}\ c))\ (a, (\mathit{comp}\ y\ r\ c, r), \mathit{set}\ r\ (\mathit{HAN}\ h)\ m) \\
= & \quad \{\ \text{define: } \mathit{exec}\ (\mathit{MARK}\ r\ c'\ c)\ (a, h, m) = \mathit{exec}\ c\ (a, (c', r), \mathit{set}\ r\ (\mathit{HAN}\ h)\ m)\ \} \\
& \mathit{exec}\ (\mathit{MARK}\ r\ (\mathit{comp}\ y\ r\ c)\ (\mathit{comp}\ x\ (\mathit{next}\ r)\ (\mathit{UNMARK}\ c))\ (a, h, m)
\end{array}
$$

That is, executing the code $\mathit{MARK}\ r\ c'\ c$ proceeds by saving the current handler in register r, and then executing the code c with the handler replaced by the new pair (c', r), a process known as 'marking' the handler stack. In this manner, the mark and unmark operations together delimit the scope of a handler to a particular segment of code.

Top-level compiler. We complete the development by considering the top-level function $\mathit{compile} :: \mathit{Expr} \to \mathit{Code}$. For arithmetic expressions, the desired behaviour was captured by $\mathit{exec}\ (\mathit{compile}\ e)\ (a, \mathit{empty}) \sqsupseteq (\mathit{eval}\ e, \mathit{empty})$ (specification 1). Based upon our experience with *comp*, in the presence of exceptions, we refine this inequation as follows:

Specification 4 (compiler correctness)**.**

$$
\begin{array}{lll}
\mathit{exec}\ (\mathit{compile}\ e)\ (a, h, \mathit{empty}) & \sqsupseteq & \textbf{case}\ \mathit{eval}\ e\ \textbf{of} \\
& & \quad \mathit{Just}\ n \quad \to (n, h, \mathit{empty}) \\
& & \quad \mathit{Nothing} \to \mathit{fail}\ h\ \mathit{empty}
\end{array}
$$

There is no need to use induction in this case as simple calculation suffices, during which we introduce a new constructor $\mathit{HALT} :: \mathit{Code}$ to transform the term being manipulated into the required form so that specification 3 can then be applied:

$$
\begin{array}{ll}
& \textbf{case}\ \mathit{eval}\ e\ \textbf{of} \\
& \quad \mathit{Just}\ n \quad \to (n, h, \mathit{empty}) \\
& \quad \mathit{Nothing} \to \mathit{fail}\ h\ \mathit{empty} \\
= & \quad \{\ \text{define } \mathit{exec}\ \mathit{HALT}\ (a, h, m) = (a, h, m)\ \} \\
& \textbf{case}\ \mathit{eval}\ e\ \textbf{of} \\
& \quad \mathit{Just}\ n \quad \to \mathit{exec}\ \mathit{HALT}\ (n, h, \mathit{empty}) \\
& \quad \mathit{Nothing} \to \mathit{fail}\ h\ \mathit{empty} \\
\sqsubseteq & \quad \{\ \text{specification 3, property 1}\ \} \\
& \mathit{exec}\ (\mathit{comp}\ e\ \mathit{first}\ \mathit{HALT})\ (a, h, \mathit{empty})
\end{array}
$$

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

The final term now has the form $exec\ c\ (a, h, empty)$, where $c = comp\ e\ first\ HALT$, from which we conclude that the following definition satisfies specification 3:

$$compile\ e = comp\ e\ first\ HALT$$

Result. In summary, we have calculated the following definitions.

Target language:

$$\begin{aligned}&\textbf{data}\ Code = HALT \mid LOAD\ Int\ Code \mid STORE\ Reg\ Code \mid ADD\ Reg\ Code \mid\\ &\qquad\qquad THROW \mid MARK\ Reg\ Code\ Code \mid UNMARK\ Code\end{aligned}$$

Compiler:

$$\begin{aligned}&compile :: Expr \to Code\\ &compile\ e = comp\ e\ first\ HALT\\ &comp :: Expr \to Reg \to Code \to Code\\ &comp\ (Val\ n)\quad r\ c = LOAD\ n\ c\\ &comp\ (Add\ x\ y)\quad r\ c = comp\ x\ r\ (STORE\ r\ (comp\ y\ (next\ r)\ (ADD\ r\ c)))\\ &comp\ (Throw)\quad r\ c = THROW\\ &comp\ (Catch\ x\ y)\ r\ c = MARK\ r\ (comp\ y\ r\ c)\ (comp\ x\ (next\ r)\ (UNMARK\ c))\end{aligned}$$

Virtual machine:

$$\begin{aligned}&\textbf{type}\ Conf = (Acc, Han, Mem)\\ &\textbf{type}\ Acc = Int\\ &\textbf{type}\ Han = (Code, Reg)\\ &\textbf{data}\ Val = VAL\ \{val :: Int\} \mid HAN\ \{han :: Han\}\end{aligned}$$

$$\begin{aligned}&exec :: Code \to Conf \to Conf\\ &exec\ HALT\ (a, h, m) = (a, h, m)\\ &exec\ (LOAD\ n\ c)\ (a, h, m) = exec\ c\ (n, h, m)\\ &exec\ (STORE\ r\ c)\ (a, h, m) = exec\ c\ (a, h, set\ r\ (VAL\ a)\ m)\\ &exec\ (ADD\ r\ c)\ (a, h, m) = exec\ c\ (val\ (get\ r\ m) + a, h, m)\\ &exec\ (THROW)\ (a, h, m) = fail\ h\ m\\ &exec\ (MARK\ r\ c'\ c)\ (a, h, m) = exec\ c\ (a, (c', r), set\ r\ (HAN\ h)\ m)\\ &exec\ (UNMARK\ c)\ (a, (\_, r), m) = exec\ c\ (a, han\ (get\ r\ m), m)\end{aligned}$$

$$\begin{aligned}&fail :: Han \to Mem \to Conf\\ &fail\ (c, r)\ m = exec\ c\ (0, han\ (get\ r\ m), m)\end{aligned}$$

In a similar manner to our first example, it is straightforward now to prove that the above definitions for *exec* and *fail* are monotonic, as required by properties 8 and 9.

4.4 Example

Consider the expression $Catch\ (Add\ (Val\ 2)\ Throw)\ (Val\ 3)$ that attempts to perform an addition that throws an exception in its second argument, which is handled by returning the value 3. Compiling this expression gives the following code sequence:

$$MARK\ 0\ (LOAD\ 3\ HALT)\ (LOAD\ 2\ (STORE\ 1\ THROW))$$

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

In turn, executing this code from an initial configuration in which the accumulator contains an arbitrary value a, the handler is set to an arbitrary code and register pair (h, r), and all registers in the memory are empty proceeds as follows, where the columns show the contents of the relevant components of the configuration after each operation:

	Acc	*Han*	*Reg* 0	*Reg* 1
	a	(h, r)	–	–
MARK 0 (*LOAD* 3 *HALT*)	a	(*LOAD* 3 *HALT*, 0)	*HAN* (h, r)	–
LOAD 2	2	(*LOAD* 3 *HALT*, 0)	*HAN* (h, r)	–
STORE 1	2	(*LOAD* 3 *HALT*, 0)	*HAN* (h, r)	*VAL* 2
THROW	0	(h, r)	*HAN* (h, r)	*VAL* 2
LOAD 3	3	(h, r)	*HAN* (h, r)	*VAL* 2
HALT	3	(h, r)	*HAN* (h, r)	*VAL* 2

That is, the expression is evaluated by first saving the current handler (h, r) in register 0 and replacing it by the new handler (*LOAD* 3 *HALT*, 0), then storing the value 2 in register 1, after which an exception is thrown. In turn, the exception results in the current handler code *LOAD* 3 *HALT* being executed and the original handler (h, r) being restored from register 0, leaving the final value 3 in the accumulator, as expected.

The above compiler can be viewed as more efficient version of the stack-based compiler in Bahr & Hutton (2015). In the event of an exception, the stack machine uses a process of 'stack unwinding' to both find the current handler and to free the memory that has become unused because of the exception. By contrast, the register machine has direct access to the handler in the configuration and does not need to explicitly free the unused memory.

4.5 Reflection

Uncaught exceptions. In our top-level statement of compiler correctness (specification 4), we are free to choose the initial handler component h of the machine, and this choice determines how the machine handles uncaught exceptions. For example, we could give h an initial value (*CRASH*, *first*), where *CRASH* is a new code constructor that simply crashes the machine by virtue of having no defining equation within the function *exec*. This choice would cause the function *fail* and hence the machine to crash in the case of an uncaught exception, either because the register *first* does not contain a valid handler or because *CRASH* is executed. Alternatively, for a more graceful treatment, we could refine the virtual machine to have an explicit representation of uncaught exceptions to avoid crashing if this case arises. In principle, however, we are free to deal with uncaught exceptions in any way that we choose, because the calculation does not depend on this.

Scalability. The ability to calculate a compiler for a language with exceptions illustrates the scalability of our methodology. In previous work (Hutton & Bahr, 2017), we introduced a compiler calculation methodology that relied on the target machine cleaning up after itself, by explicitly freeing up registers that were no longer being used. However, this approach only works if we have a language with a control flow that is known at compile time. If the control flow is not known until run time, as is the case in the presence of

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

exceptions, then it is not clear how the machine should clean up after itself. In particular, it is not clear how this can be achieved in a systematic way by means of calculation.

Note: strictly speaking, the control flow in our language is known at compile time, as there is no primitive for reading input from the user. However, our compiler calculation does not rely on this and can readily be extended with such a primitive.

Partiality. By avoiding the explicit freeing of registers, and instead using implicit freeing by means of the ordering $\sqsupseteq$ on memories, we are able to use the same methodology that we developed for stack machines (Bahr & Hutton, 2015). In particular, we begin with a partial specification of compiler correctness in terms of some components that are initially undefined or incomplete. The extension of the machine configuration with an additional component to handle exceptions is new but follows the methodology of Bahr & Hutton (2015) in that it extends the configuration type to handle new computational behaviour, in this case exceptions, but it does so using a partial specification.

Linearity. The compiler produces code that is not fully linear, because the *MARK* operation takes two *Code* arguments. This branching structure corresponds to the branching in control flow that is inherent in the semantics of *Catch*. However, if desired, a compiler that produces linear code can be obtained by modifying *MARK* to take a code pointer as its second argument rather than code itself (Hutton & Wright, 2006; Bahr, 2014).

5 Lambda calculus

For our final example, we extend the language of arithmetic expressions with support for variables, abstraction and application in a call-by-value lambda calculus:

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

Informally, $\mathit{Var}\ i$ is the variable with de Bruijn index $i \geqslant 0$, while $\mathit{Abs}\ x$ constructs an abstraction over expression x and $\mathit{App}\ x\ y$ applies the abstraction that results from evaluating expression x to the value of expression y. For example, the function $\lambda n \to n + 1$ that increments an integer value can be represented as follows:

$$\begin{aligned}&\mathit{inc} :: \mathit{Expr}\\ &\mathit{inc} = \mathit{Abs}\ (\mathit{Add}\ (\mathit{Var}\ 0)\ (\mathit{Val}\ 1))\end{aligned}$$

Because the result of evaluating an expression may now be a function, we extend the value domain for the semantics to include functional values:

$$\textbf{data}\ \mathit{Value} = \mathit{Num}\ \mathit{Int} \mid \mathit{Fun}\ (\mathit{Value} \to \mathit{Value})$$

To interpret free variables the semantics also requires an *environment*, which can be represented simply as a list of values, with the value of the variable with de Bruijn index i given by indexing into the list at this position, written as $e\ !!\ i$:

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

Using these ideas, the semantics for the language can now be defined as a function that evaluates an expression to a value in a given environment:

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Expr} \to \mathit{Env} \to \mathit{Value}\\
&\mathit{eval}\ (\mathit{Val}\ n)\quad e = \mathit{Num}\ n\\
&\mathit{eval}\ (\mathit{Add}\ x\ y)\ e = \mathbf{case}\ \mathit{eval}\ x\ e\ \mathbf{of}\\
&\qquad\qquad\qquad\qquad \mathit{Num}\ n \to \mathbf{case}\ \mathit{eval}\ y\ e\ \mathbf{of}\\
&\qquad\qquad\qquad\qquad\quad \mathit{Num}\ m \to \mathit{Num}\ (n + m)\\
&\mathit{eval}\ (\mathit{Var}\ i)\quad e = e\ !!\ i\\
&\mathit{eval}\ (\mathit{Abs}\ x)\quad e = \mathit{Fun}\ (\lambda v \to \mathit{eval}\ x\ (v : e))\\
&\mathit{eval}\ (\mathit{App}\ x\ y)\ e = \mathbf{case}\ \mathit{eval}\ x\ e\ \mathbf{of}\\
&\qquad\qquad\qquad\qquad \mathit{Fun}\ f \to f\ (\mathit{eval}\ y\ e)
\end{aligned}
$$

For example, applying the function *eval* to the expression $\mathit{App}\ \mathit{inc}\ (\mathit{Val}\ 1)$ and the empty environment $[\,]$ gives the result $\mathit{Num}\ 2$, as expected. Note that *eval* is now a partial function, because expressions in our lambda calculus language may be ill-formed or non-terminating. We will return to the issue of partiality at the end of this example.

Our goal now is to calculate a compiler for the above language, building on our experience with exceptions. Our development proceeds in four steps:

1. Refine the semantics for the language;
2. Refine the underlying machine model;
3. Refine the specification of compiler correctness;
4. Calculate a compiler that satisfies the specification.

5.1 Language semantics

We could now proceed to calculate a compiler using the above semantics. However, the calculation would get stuck in the abstraction case, due to the fact that the semantics is now higher order by virtue of abstractions denoting functions. However, as observed in our previous work (Bahr & Hutton, 2015), this problem can be resolved by transforming the semantics into first-order form using *defunctionalisation* (Reynolds, 1972), and reformulating the resulting evaluation function as a *big-step operational* (or natural) semantics. The latter step allows the compiler calculation to be performed using rule induction rather than using structural induction, which is no longer applicable because defunctionalisation gives rise to an evaluator that is no longer structurally recursive.

Defunctionalising the semantics is a purely mechanical process, as described in Bahr & Hutton (2015), and results in a new first-order version of the *Value* type:

$$\mathbf{data}\ \mathit{Value} = \mathit{Num}\ \mathit{Int} \mid \mathit{Clo}\ \mathit{Expr}\ \mathit{Env}$$

The new constructor *Clo* corresponds to a *closure*, which comprises an expression and an environment that captures its free variables. In turn, the resulting evaluation function is reformulated as a big-step operational semantics, where we write $x \Downarrow_e v$ to mean that the expression x can evaluate to the value v within the environment e. Formally, the evaluation relation $\Downarrow \subseteq \mathit{Expr} \times \mathit{Env} \times \mathit{Value}$ is defined by the following inference rules:

$$\frac{}{\mathit{Val}\ n \Downarrow_e \mathit{Num}\ n} \qquad \frac{x \Downarrow_e \mathit{Num}\ n \qquad y \Downarrow_e \mathit{Num}\ m}{\mathit{Add}\ x\ y \Downarrow_e \mathit{Num}\ (n + m)} \qquad \frac{e\ !!\ i \text{ is defined}}{\mathit{Var}\ i \Downarrow_e e\ !!\ i}$$

$$\frac{}{\mathit{Abs}\ x \Downarrow_e \mathit{Clo}\ x\ e} \qquad \frac{x \Downarrow_e \mathit{Clo}\ x'\ e' \qquad y \Downarrow_e v \qquad x' \Downarrow_{v:e'} w}{\mathit{App}\ x\ y \Downarrow_e w}$$

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

5.2 Machine model

We now refine our machine model. First of all, because the value domain for the semantics is now *Value* rather than simply *Int*, we modify the accumulator type similarly:

type $Acc = Value$

Secondly, as with the exceptions example, we assume that registers in the memory store values of a new type *Val*, which initially only contains integer values, but will be extended as necessary during the calculation process. That is, we have:

data $Val = VAL\ Int$

$set :: Reg \to Val \to Mem \to Mem$

$get :: Reg \to Mem \to Val$

In turn, just as we added a new component *Han* to the configuration type to handle exceptions, so we add a new, as yet undefined, component *Lam* to manage any additional aspects that are required to implement the lambda calculus:

type $Conf = (Acc, Lam, Mem)$

type $Lam = \cdots$

And finally, one of the guiding principles from our previous work (Bahr & Hutton, 2015) is that additional data structures on which the semantics for the source language depends should be added to the type of machine configurations. Hence, in the case of the lambda calculus, we further extend the configuration type with an environment:

type $Conf = (Acc, Env, Lam, Mem)$

5.3 Compiler correctness

Using the above definitions, we can now specify the desired behaviour of the compilation function $comp :: Expr \to Reg \to Code \to Code$. The resulting specification is similar to the original version for arithmetic expressions, except that the semantics is now defined as an evaluation relation $\Downarrow \subseteq Expr \times Env \times Value$, and the machine $exec :: Code \to Conf \to Conf$ now operates on configurations that have four components:

$$freeFrom\ r\ m \ \wedge \ x \Downarrow_e v \ \Rightarrow \ exec\ (comp\ x\ r\ c)\ (a, e, l, m) \ \sqsupseteq \ exec\ c\ (v, e, l, m)$$

It is possible to calculate a compiler from the above specification, but the resulting machine would not be satisfactory because closures contain unevaluated expressions, whereas we normally expect all source expressions to be compiled away. As in Bahr & Hutton (2015), the solution is simply to replace the expression component within a closure by compiled code for the expression, by means of the following new type definitions:

data $Value' = Num'\ Int \mid Clo'\ Code\ Env'$

type $Env' = [Value']$

In this manner, *Value* corresponds to semantic values, while *Value′* corresponds to machine values. These new types are in turn used to redefine the other basic types:

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

$$\textbf{type}\ \mathit{Conf} = (\mathit{Acc}, \mathit{Env}', \mathit{Lam}, \mathit{Mem})$$
$$\textbf{type}\ \mathit{Acc} \quad = \mathit{Value}'$$

The changes to these definitions means that the specification for *comp* is no longer type correct, because the relation $\Downarrow$ and function *exec* operate on different value types, namely *Value* and *Value*′. We therefore need a conversion function between the two types. The case for *Num* is trivial, while we leave the case for *Clo* undefined at present and aim to derive a definition for this case during the calculation process:

$$\begin{aligned}&\mathit{conv} :: \mathit{Value} \to \mathit{Value}'\\ &\mathit{conv}\ (\mathit{Num}\ n) = \mathit{Num}'\ n\\ &\mathit{conv}\ (\mathit{Clo}\ x\ e) = \cdots\end{aligned}$$

We lift *conv* to environments by applying the function to each value in the list:

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

Using the above ideas, we can now modify the specification for the compilation function *comp* to include the necessary type conversions:

Specification 5 (generalised compiler correctness)**.**

$$\begin{aligned}&\mathit{freeFrom}\ r\ m \ \wedge\ x \Downarrow_e v\\ \Rightarrow\ &\\ &\mathit{exec}\ (\mathit{comp}\ x\ r\ c)\ (a, \mathit{conv}_{\mathsf{E}}\ e, l, m) \sqsupseteq \mathit{exec}\ c\ (\mathit{conv}\ v, \mathit{conv}_{\mathsf{E}}\ e, l, m)\end{aligned}$$

5.4 Compiler calculation

Using specification 5, we now calculate definitions for *comp* and *exec* by constructive rule induction on the assumption $x \Downarrow_e v$. In each case, we aim to rewrite the right-hand side $\mathit{exec}\ c\ (\mathit{conv}\ v, \mathit{conv}_{\mathsf{E}}\ e, l, m)$ of the inequation into the form $\mathit{exec}\ c'\ (a, \mathit{conv}_{\mathsf{E}}\ e, l, m)$ for some code c', from which we can then conclude that the definition $\mathit{comp}\ x\ r\ c = c'$ satisfies the specification in this case. As with the previous example, during this process we will add new constructors to *Code* and *Val*, and new equations for *exec*. Along the way, we will also define the new type *Lam* and complete the definition of *conv*.

Case: *Val n* The case for $\mathit{Val}\ n \Downarrow_e \mathit{Num}\ n$ is straightforward:

$$\begin{aligned}&\mathit{exec}\ c\ (\mathit{conv}\ (\mathit{Num}\ n), \mathit{conv}_{\mathsf{E}}\ e, l, m)\\ =\quad &\{\ \text{definition of } \mathit{conv}\ \}\\ &\mathit{exec}\ c\ (\mathit{Num}'\ n, \mathit{conv}_{\mathsf{E}}\ e, l, m)\\ =\quad &\{\ \text{define: } \mathit{exec}\ (\mathit{LOAD}\ n\ c)\ (a, e, l, m) = \mathit{exec}\ c\ (\mathit{Num}'\ n, e, l, m)\ \}\\ &\mathit{exec}\ (\mathit{LOAD}\ n\ c)\ (a, \mathit{conv}_{\mathsf{E}}\ e, l, m)\end{aligned}$$

Case: *Var i* The case for $\mathit{Var}\ i \Downarrow_e e\ !!\ i$ is also straightforward:

$$\begin{aligned}&\mathit{exec}\ c\ (\mathit{conv}\ (e\ !!\ i), \mathit{conv}_{\mathsf{E}}\ e, l, m)\\ =\quad &\{\ \text{indexing lemma}\ \}\end{aligned}$$

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

$$
\begin{array}{ll}
 & exec\ c\ (map\ conv\ e\ !!\ i, conv_{\mathsf{E}}\ e, l, m) \\
= & \quad \{\ \text{definition of } conv_{\mathsf{E}}\ \} \\
 & exec\ c\ (conv_{\mathsf{E}}\ e\ !!\ i, conv_{\mathsf{E}}\ e, l, m) \\
= & \quad \{\ \text{define: } exec\ (LOOKUP\ i\ c)\ (a, e, l, m) = exec\ c\ (e\ !!\ i, e, l, m)\ \} \\
 & exec\ (LOOKUP\ i\ c)\ (a, conv_{\mathsf{E}}\ e, l, m)
\end{array}
$$

The indexing lemma used above states that $f\ (xs\ !!\ i) = (map\ f\ xs)\ !!\ i$ for any strict f, and allows us to generalise over $conv_{\mathsf{E}}\ e$ when defining *exec* for *LOOKUP*. The function *conv* is strict because it is defined by pattern matching on its argument.

Case: *Add x y* In the case for $Add\ x\ y \Downarrow_e Num\ (n + n')$, we can assume $x \Downarrow_e Num\ n$ and $y \Downarrow_e Num\ n'$ by the inference rule for *Add*, and induction hypotheses for the expressions x and y. The calculation proceeds in the same way as for simple arithmetic expressions, except that we now need to apply the necessary conversion functions:

$$
\begin{array}{ll}
 & exec\ c\ (conv\ (Num\ (n + n')), conv_{\mathsf{E}}\ e, l, m) \\
= & \quad \{\ \text{definition of } conv\ \} \\
 & exec\ c\ (Num'\ (n + n'), conv_{\mathsf{E}}\ e, l, m) \\
\sqsubseteq & \quad \{\ \text{Properties 3 \& 8}\ \} \\
 & exec\ c\ (Num'\ (n + n'), conv_{\mathsf{E}}\ e, l, set\ r\ (VAL\ n)\ m) \\
= & \quad \{\ \text{Property 2, where } VAL\ n'' = get\ r\ (set\ r\ (VAL\ n)\ m)\ \} \\
 & exec\ c\ (Num'\ (n'' + n'), conv_{\mathsf{E}}\ e, l, set\ r\ (VAL\ n)\ m) \\
= & \quad \left\{ \begin{array}{l} \text{define: } exec\ (ADD\ r\ c)\ (Num'\ a, e, l, m) = exec\ c\ (Num'\ (n + a), e, l, m) \\ \quad \textbf{where } VAL\ n = get\ r\ m \end{array} \right\} \\
 & exec\ (ADD\ r\ c)\ (Num'\ n', conv_{\mathsf{E}}\ e, l, set\ r\ (VAL\ n)\ m) \\
= & \quad \{\ \text{definition of } conv\ \} \\
 & exec\ (ADD\ r\ c)\ (conv\ (Num\ n'), conv_{\mathsf{E}}\ e, l, set\ r\ (VAL\ n)\ m) \\
\sqsubseteq & \quad \{\ \text{induction hypothesis for } y\ \} \\
 & exec\ (comp\ y\ (next\ r)\ (ADD\ r\ c))\ (Num'\ n, conv_{\mathsf{E}}\ e, l, set\ r\ (VAL\ n)\ m) \\
= & \quad \{\ \text{define: } exec\ (STORE\ r\ c)\ (Num'\ n, e, l, m) = exec\ c\ (Num'\ n, e, l, set\ r\ (VAL\ n)\ m)\ \} \\
 & exec\ (STORE\ r\ (comp\ y\ (next\ r)\ (ADD\ r\ c)))\ (Num'\ n, conv_{\mathsf{E}}\ e, l, m) \\
\sqsubseteq & \quad \{\ \text{induction hypothesis for } x\ \} \\
 & exec\ (comp\ x\ r\ (STORE\ r\ (comp\ y\ (next\ r)\ (ADD\ r\ c))))\ (a, conv_{\mathsf{E}}\ e, l, m)
\end{array}
$$

Case: *App x y* In the case for $App\ x\ y \Downarrow_e w$, we can assume $x \Downarrow_e Clo\ x'\ e'$, $y \Downarrow_e v$, and $x' \Downarrow_{v:e'} w$ by the inference rule for $App\ x\ y$, together with induction hypotheses for x, y and x'. The calculation then proceeds in the familiar way, by adding code and value constructors to bring the configuration into the right form to apply the induction hypotheses. First of all, to apply the induction hypothesis for x', we save and restore an environment in a register using a new value constructor *ENV* and code constructor *RET*:

$$
\begin{array}{ll}
 & exec\ c\ (conv\ w, conv_{\mathsf{E}}\ e, l, m) \\
\sqsubseteq & \quad \{\ \text{Property 3 \& 8}\ \} \\
 & exec\ c\ (conv\ w, conv_{\mathsf{E}}\ e, l, set\ r\ (ENV\ (conv_{\mathsf{E}}\ e))\ m) \\
= & \quad \{\ \text{Property 2, where } ENV\ e'' = get\ r\ (set\ r\ (ENV\ (conv_{\mathsf{E}}\ e))\ m)\ \} \\
 & exec\ c\ (conv\ w, e'', l, set\ r\ (ENV\ (conv_{\mathsf{E}}\ e))\ m)
\end{array}
$$

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

$$
\begin{array}{ll}
= & \left\{ \begin{array}{l} \text{define: } exec\,(RET\ r\ c)\,(a, e', l, m) = exec\ c\,(a, e, l, m) \\ \quad \textbf{where } ENV\ e = get\ r\ m \end{array} \right\} \\
& exec\,(RET\ r\ c)\,(conv\ w, conv\ v : conv_{\mathsf{E}}\ e', l, set\ r\,(ENV\,(conv_{\mathsf{E}}\ e))\ m) \\
= & \quad \{\ \text{induction hypothesis for } x'\ \} \\
& exec\,(comp\ x'\ (next\ r)\,(RET\ r\ c))\ (\ a'\ , conv\ v : conv_{\mathsf{E}}\ e', l, set\ r\,(ENV\,(conv_{\mathsf{E}}\ e))\ m)
\end{array}
$$

Unfortunately, if we continued the calculation from here we would get stuck. In particular, we would end up with a term of the form $exec\ c'\ (a, conv_{\mathsf{E}}\ e, l, m)$, where the code c' contains the subterm $comp\ x'\ (next\ r)\,(RET\ r\ c)$ that was introduced above. This is problematic, because we cannot simply define $comp\,(App\ x\ y)\ r\ c = c'$ if the code c' contains the free variable x'. The alternative is to assume that the code $comp\ x'\ (next\ r)\,(RET\ r\ c)$ comes from the accumulator and use the induction hypothesis for $x \Downarrow_e Clo\ x'\ e'$ to discharge this assumption. However, following this alternative is also problematic, because we cannot recover the two arguments r and c of RET. Fortunately, there is a simple solution that allows the calculation to proceed: we revise our earlier choice for the behaviour of RET so that it no longer takes the register r and code c as arguments.

To avoid taking c as an argument, the operation RET can take it from the memory instead. We could attempt to do the same for r, but this would not resolve the problem because $next\ r$ is also used as an argument to $comp$. However, we can eliminate the need for the variable r by replacing it with a fixed register, for which the natural choice is the *first* register in the memory. In order to use *first* for this purpose, we then need to manipulate the memory m to be *empty*, to ensure that the *freeFrom* side condition for the induction hypothesis is satisfied by appealing to property 1 (empty memory).

In summary, we can repair the above calculation by storing the code c in memory, in addition to the environment $conv_{\mathsf{E}}\ e$, and using the *empty* memory in place of m. The first change can be realised by replacing the value constructor ENV by a more general constructor CLO that takes both code and an environment as arguments. For the second, we then need to store the original memory m somewhere so that it can be restored later on, as we are aiming to rewrite the term into the form $exec\ c'\ (a, conv_{\mathsf{E}}\ e, l, m)$. It cannot be stored in the memory itself, because our goal is to obtain the empty memory. Thus, the only reasonable choice is to store m in the l component of the configuration. Because l itself must also be restored, we define the corresponding type Lam as a list of memories, so that the cons operator $(:) :: Mem \to Lam \to Lam$ then allows us to store both m and l:

$$\textbf{type } Lam = [Mem]$$

Using the above ideas, we restart the calculation for the App case:

$$
\begin{array}{ll}
& exec\ c\,(conv\ w, conv_{\mathsf{E}}\ e, l, m) \\
= & \quad \{\ \text{Property 2, where } CLO\ c'\ e'' = get\ first\,(set\ first\,(CLO\ c\,(conv_{\mathsf{E}}\ e))\ empty)\ \} \\
& exec\ c'\,(conv\ w, e'', l, m) \\
= & \left\{ \begin{array}{l} \text{define: } exec\ RET\,(a, e', m : l, m') = exec\ c\,(a, e, l, m) \\ \quad \textbf{where } CLO\ c\ e = get\ first\ m' \end{array} \right\} \\
& exec\ RET\,(conv\ w, conv\ v : conv_{\mathsf{E}}\ e', m : l, set\ first\,(CLO\ c\,(conv_{\mathsf{E}}\ e))\ empty) \\
\sqsubseteq & \quad \{\ \text{induction hypothesis for } x' \Downarrow_{v:e'} w\ \} \\
& exec\,(comp\ x'\ (next\ first)\ RET) \\
& \quad (\ a'\ , conv\ v : conv_{\mathsf{E}}\ e', m : l, set\ first\,(CLO\ c\,(conv_{\mathsf{E}}\ e))\ empty)
\end{array}
$$

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

Next, we aim to apply the induction hypothesis for $y \Downarrow_e v$. For this to be applicable, the term being manipulated needs to be of the form $exec\ c'\ (conv\ v, conv_{\mathsf{E}}\ e, l', m')$, which means that we need to solve the following inequation for some c', l', m' and a':

$$\begin{array}{l} exec\ \boxed{c'}\ (conv\ v, conv_{\mathsf{E}}\ e, \boxed{l'}, \boxed{m'}) \sqsupseteq exec\ (comp\ x'\ (next\ first)\ RET) \\ \quad (\boxed{a'}, conv\ v : conv_{\mathsf{E}}\ e', m : l, set\ first\ (CLO\ c\ (conv_{\mathsf{E}}\ e))\ empty) \end{array}$$

As usual, to solve this inequation, we introduce a new equation for *exec*. To this end, we need to determine from which of the existentially quantified variables c', l' and m' on the left-hand side of the inequation we are going to retrieve the universally quantified elements on the right-hand side. We consider each of these elements in turn:

- For the code $comp\ x'\ (next\ first)\ RET$, the simplest option is to store it in c'. However, as observed above, this will result in the calculation getting stuck later on. Hence, we choose the other option and store this code fragment in m'.
- For the accumulator a', we have free choice as this variable is existentially quantified. We choose $a' = conv\ v$ so that it matches the left-hand side.
- For the environment $conv_{\mathsf{E}}\ e'$, we have the choice of putting it into either m' or c'. We choose to put it into the memory m', as putting it into the code c' would result in a run-time value being stored in a compile-time value.
- For the memory m, the natural choice is to store it in m' by requiring $m' \sqsupseteq m$, while for the value l, the simplest option is to equate it with l' by taking take $l = l'$.
- Finally, for the code c, the natural choice is to store it in c'.

How should we satisfy the above requirements? First of all, in order to store both the code $comp\ x'\ (next\ first)\ RET$ and the environment $conv_{\mathsf{E}}\ e'$ in the memory m', we can use the new value constructor *CLO* that we have already introduced above. We then require $get\ r'\ m' = CLO\ (comp\ x'\ (next\ first)\ RET)\ (conv_{\mathsf{E}}\ e')$ for some register r'. As we also need to satisfy the memory requirement $m' \sqsupseteq m$, we thus define

$$m' = set\ \boxed{r'}\ (CLO\ (comp\ x'\ (next\ first)\ RET)\ (conv_{\mathsf{E}}\ e'))\ m$$

and once again appeal to properties 2 and 3. In order to use the latter of these properties, we need to take $r' = r$. Finally, we also need to store the first free register r somewhere. The easiest choice is to store it the code c' along with the code c, by means of a new code constructor *APP* that takes both a register and code as arguments.

Using the above ideas, we now resume the calculation for the *App* case, proceeding in a few steps to the application of the induction hypothesis for $y \Downarrow_e v$:

$$\begin{array}{ll} & exec\ (comp\ x'\ (next\ first)\ RET) \\ & \quad (conv\ v, conv\ v : conv_{\mathsf{E}}\ e', m : l, set\ first\ (CLO\ c\ (conv_{\mathsf{E}}\ e))\ empty) \\ \sqsubseteq & \quad \{\ \text{Properties 3 and 8}\ \} \\ & exec\ (comp\ x'\ (next\ first)\ RET) \\ & \quad (conv\ v, conv\ v : conv_{\mathsf{E}}\ e', set\ r\ (CLO\ (comp\ x'\ (next\ first)\ RET)\ (conv_{\mathsf{E}}\ e'))\ m : l, \\ & \qquad set\ first\ (CLO\ c\ (conv_{\mathsf{E}}\ e))\ empty) \end{array}$$

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

$$
\begin{array}{ll}
 & \left\{ \begin{array}{l} \text{define: } exec\ (APP\ r\ c)\ (a, e, l, m) = \\ \quad exec\ c'\ (a, a : e', m : l, set\ first\ (CLO\ c\ e)\ empty) \\ \quad\quad \textbf{where}\ CLO\ c'\ e' = get\ r\ m \end{array} \right\} \\
 & exec\ (APP\ r\ c) \\
 & \quad (conv\ v, conv_{\mathsf{E}}\ e, l, set\ r\ (CLO\ (comp\ x'\ (next\ first)\ RET)\ (conv_{\mathsf{E}}\ e'))\ m) \\
\sqsubseteq & \quad \{\ \text{induction hypothesis for } y \Downarrow_e v\ \} \\
 & exec\ (comp\ y\ (next\ r)\ (APP\ r\ c)) \\
 & \quad (a', conv_{\mathsf{E}}\ e, l, set\ r\ (CLO\ (comp\ x'\ (next\ first)\ RET)\ (conv_{\mathsf{E}}\ e'))\ m)
\end{array}
$$

Finally, we aim to apply the induction hypothesis for $x \Downarrow_e Clo\ x'\ e'$. That is, our goal is to solve the following inequation for some c', l', m' and a':

$$
\begin{array}{l}
exec\ c'\ (conv\ (Clo\ x'\ e'), conv_{\mathsf{E}}\ e, l', m') \sqsupseteq exec\ (comp\ y\ (next\ r)\ (APP\ r\ c)) \\
\qquad (a', conv_{\mathsf{E}}\ e, l, set\ r\ (CLO\ (comp\ x'\ (next\ first)\ RET)\ (conv_{\mathsf{E}}\ e'))\ m)
\end{array}
$$

As we observed in our initial calculation attempt, we cannot simply store the code fragment $comp\ x'\ (next\ first)\ RET$ in c'. This choice would lead into a dead end, because when compiling $App\ x\ y$ the compiler does not have access to the expression x'. Similarly, it is not immediately clear from where to retrieve $conv_{\mathsf{E}}\ e'$. However, we can freely define $conv\ (Clo\ x'\ e')$ in order that it gives us the required data:

$$conv\ (Clo\ x'\ e') = Clo'\ (comp\ x'\ (next\ first)\ RET)\ (conv_{\mathsf{E}}\ e')$$

Note that this definition makes use of the fact that RET does not take a register and code as arguments, unlike in our initial calculation attempt that became stuck.

The remaining elements are stored in the usual way. In particular, we store the code $comp\ y\ (next\ r)\ (APP\ r\ c)$ in c', the memory m in m' by taking $m = m'$, the lambda component l in l' by taking $l = l'$, and finally, we choose $a' = conv\ (Clo\ x'\ e')$ to match the left-hand side. We can then conclude the App calculation as follows:

$$
\begin{array}{ll}
 & exec\ (comp\ y\ (next\ r)\ (APP\ r\ c))\ (conv\ (Clo\ x'\ e'), conv_{\mathsf{E}}\ e, l, \\
 & \qquad set\ r\ (CLO\ (comp\ x'\ (next\ first)\ RET)\ (conv_{\mathsf{E}}\ e'))\ m) \\
= & \left\{ \begin{array}{l} \text{define: } exec\ (STC\ r\ c)\ (Clo'\ c'\ e', e, l, m) = \\ \quad exec\ c\ (Clo'\ c'\ e', e, l, set\ r\ (CLO\ c'\ e')\ m) \end{array} \right\} \\
 & exec\ (STC\ r\ (comp\ y\ (next\ r)\ (APP\ r\ c)))\ (conv\ (Clo\ x'\ e'), conv_{\mathsf{E}}\ e, l, m) \\
= & \quad \{\ \text{induction hypothesis for } x \Downarrow_e Clo\ x'\ e'\ \} \\
 & exec\ (comp\ x\ r\ (STC\ r\ (comp\ y\ (next\ r)\ (APP\ r\ c))))\ (a, conv_{\mathsf{E}}\ e, l, m)
\end{array}
$$

Case: *Abs x* Using the new equation for $conv$, the abstraction case simply amounts to introducing a code constructor ABS that stores a closure in the accumulator:

$$
\begin{array}{ll}
 & exec\ c\ (conv\ (Clo\ x\ e), conv_{\mathsf{E}}\ e, l, m) \\
= & \quad \{\ \text{definition of } conv\ \} \\
 & exec\ c\ (Clo'\ (comp\ x\ (next\ first)\ RET)\ (conv_{\mathsf{E}}\ e), conv_{\mathsf{E}}\ e, l, m) \\
= & \quad \{\ \text{define: } exec\ (ABS\ c'\ c)\ (a, e, l, m) = exec\ c\ (Clo'\ c'\ e, e, l, m)\ \} \\
 & exec\ (ABS\ (comp\ x\ (next\ first)\ RET)\ c)\ (a, conv_{\mathsf{E}}\ e, l, m)
\end{array}
$$

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

Result. In summary, we have calculated the definitions below. As with the previous examples, the top-level compilation function *compile* is defined by applying *comp* to the *first* register and a *HALT* constructor that returns the current configuration.

Target language:

$$\begin{aligned}&\textbf{data}\ Code = HALT \mid LOAD\ Int\ Code \mid LOOKUP\ Int\ Code\\ &\qquad\qquad \mid STORE\ Reg\ Code \mid ADD\ Reg\ Code\\ &\qquad\qquad \mid STC\ Reg\ Code \mid APP\ Reg\ Code\\ &\qquad\qquad \mid ABS\ Code\ Code \mid RET\end{aligned}$$

Compiler:

$$\begin{aligned}&compile :: Expr \to Code\\ &compile\ e = comp\ e\ first\ HALT\\ &comp :: Expr \to Reg \to Code \to Code\\ &comp\ (Val\ n)\quad r\ c = LOAD\ n\ c\\ &comp\ (Var\ i)\quad r\ c = LOOKUP\ i\ c\\ &comp\ (Add\ x\ y)\ r\ c = comp\ x\ r\ (STORE\ r\ (comp\ y\ (next\ r)\ (ADD\ r\ c)))\\ &comp\ (App\ x\ y)\ r\ c = comp\ x\ r\ (STC\ r\ (comp\ y\ (next\ r)\ (APP\ r\ c)))\\ &comp\ (Abs\ x)\quad r\ c = ABS\ (comp\ x\ (next\ first)\ RET)\ c\end{aligned}$$

Virtual machine:

$$\begin{aligned}&\textbf{type}\ Conf = (Acc, Env', Lam, Mem)\\ &\textbf{type}\ Acc = Value'\\ &\textbf{type}\ Env' = [Value']\\ &\textbf{data}\ Value' = Num'\ Int \mid Clo'\ Code\ Env'\\ &\textbf{type}\ Lam = [Mem]\\ &\textbf{data}\ Val = NUM\ Int \mid CLO\ Code\ Env'\end{aligned}$$

$$\begin{aligned}&exec :: Code \to Conf \to Conf\\ &exec\ HALT\ (a,e,l,m) = (a,e,l,m)\\ &exec\ (LOAD\ n\ c)\ (a,e,l,m) = exec\ c\ (Num'\ n,e,l,m)\\ &exec\ (LOOKUP\ i\ c)\ (a,e,l,m) = exec\ c\ (e\ !!\ i,e,l,m)\\ &exec\ (STORE\ r\ c)\ (Num'\ n,e,l,m) = exec\ c\ (Num'\ n,e,l,set\ r\ (NUM\ n)\ m)\\ &exec\ (ADD\ r\ c)\ (Num'\ a,e,l,m) = exec\ c\ (Num'\ (n+a),e,l,m)\\ &\qquad\textbf{where}\ NUM\ n = get\ r\ m\\ &exec\ (STC\ r\ c)\ (Clo'\ c'\ e',e,l,m) = exec\ c\ (Clo'\ c'\ e',e,l,set\ r\ (CLO\ c'\ e')\ m)\\ &exec\ (APP\ r\ c)\ (a,e,l,m) = exec\ c'\ (a,a:e',m:l,set\ first\ (CLO\ c\ e)\ empty)\\ &\qquad\textbf{where}\ CLO\ c'\ e' = get\ r\ m\\ &exec\ (ABS\ c'\ c)\ (a,e,l,m) = exec\ c\ (Clo'\ c'\ e,e,l,m)\\ &exec\ RET\ (a,e',m:l,m') = exec\ c\ (a,e,l,m)\\ &\qquad\textbf{where}\ CLO\ c\ e = get\ first\ m'\end{aligned}$$

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

Conversion functions:

$$
\begin{aligned}
&conv :: Value \to Value' \\
&conv\,(Num\,n) = Num'\,n \\
&conv\,(Clo\,x\,e) = Clo'\,(comp\,x\,(next\,first)\,RET)\,(conv_{\mathsf{E}}\,e) \\
&conv_{\mathsf{E}} :: Env \to Env' \\
&conv_{\mathsf{E}}\,e = map\,conv\,e
\end{aligned}
$$

We conclude this section with two remarks. First of all, in the above calculation we used property 8 (*exec* is monotonic) informally, in the sense that the memory can always be freely extended using $\sqsubseteq$. More formally, we need to extend $\sqsubseteq$ to the new form of machine configurations in order for property 8 to make sense. In contrast to the first two compiler calculations, memories can occur in more than one place, because our calculation revealed that the *Lam* component of the configuration type also contains memories. Hence, we first extend $\sqsubseteq$ pointwise to *Lam* by the following inductive definition:

$$
\begin{array}{rcl}
[\,] \sqsubseteq [\,] & \Leftrightarrow & \text{True} \\
(m : l) \sqsubseteq (m' : l') & \Leftrightarrow & m \sqsubseteq m' \;\wedge\; l \sqsubseteq l'
\end{array}
$$

This definition formally justifies the use of property 8 in the calculation for the *App* case. In turn, we use this definition to extend $\sqsubseteq$ to machine configurations:

$$
(a, e, l, m) \sqsubseteq (a', e', l', m') \quad \Leftrightarrow \quad a = a' \;\wedge\; e = e' \;\wedge\; l \sqsubseteq l' \;\wedge\; m \sqsubseteq m'
$$

And finally, note that the two types $Value'$ and *Val* are isomorphic and could be refactored into a single type. However, this property of the two types is coincidental and does not generalise. In particular, changing the source language slightly, for example by adding exceptions, would yield types $Value'$ and *Val* that are not isomorphic.

5.5 Example

Recall that the increment function $\lambda n \to n + 1$ can be written in our language as $inc = Abs\,(Add\,(Var\,0)\,(Val\,1))$. Now consider the expression $App\,inc\,(Val\,2)$, which increments the value 2 to produce the result 3. Compiling this expression gives the code

$$ABS\,c\,(STC\,0\,(LOAD\,2\,(APP\,0\,HALT)))$$

in which we abbreviate the code for the *inc* function by:

$$c = LOOKUP\,0\,(STORE\,1\,(LOAD\,1\,(ADD\,1\,RET))))$$

In turn, executing the above code from an arbitrary initial configuration (a, e, l, m) in which all registers in the memory m are empty proceeds as follows, where the memory m' contains the closure $CLO\,c\,e$ in register 0 and is otherwise empty:

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

	Acc	*Env′*	*Lam*	*Reg* 0	*Reg* 1
	a	*e*	*l*	–	–
ABS c	*Clo′ c e*	*e*	*l*	–	–
STC 0	*Clo′ c e*	*e*	*l*	*CLO c e*	–
LOAD 2	*Num′* 2	*e*	*l*	*CLO c e*	–
APP 0	*Num′* 2	*Num′* 2 : *e*	*m′* : *l*	*CLO HALT e*	–
LOOKUP 0	*Num′* 2	*Num′* 2 : *e*	*m′* : *l*	*CLO HALT e*	–
STORE 1	*Num′* 2	*Num′* 2 : *e*	*m′* : *l*	*CLO HALT e*	*VAL* 2
LOAD 1	*Num′* 1	*Num′* 2 : *e*	*m′* : *l*	*CLO HALT e*	*VAL* 2
ADD 1	*Num′* 3	*Num′* 2 : *e*	*m′* : *l*	*CLO HALT e*	*VAL* 2
RET	*Num′* 3	*e*	*l*	*CLO c e*	–
HALT	*Num′* 3	*e*	*l*	*CLO c e*	–

That is, the expression is evaluated by storing the closure comprising the *inc* function code and current environment in register 0, storing the argument value 2 in the accumulator, and then calling the function. In turn, this results in the code for increment being executed with the argument 2 supplied as first value in the environment, with the current memory being saved in the lambda component, after which the previous components of the configuration are restored, leaving the result 3 in the accumulator.

The above approach to compiling lambda expressions can be viewed as a lower-level version of the stack-based approach from Bahr & Hutton (2015), in which the use of relative addressing to a stack is replaced by direct addressing to registers. The extension of the machine configuration with a *Lam* component to handle lambda expressions is new, but as with the exceptions example follows the methodology of Bahr & Hutton (2015), in that it extends the configuration type to handle new computational behaviour. The resulting machine is also similar to Landin's *SECD* machine (1964), where *E* corresponds to the environment *Env′*, *C* to *Code* and *D* to the *Lam* component, with the key difference being that our machine utilises registers rather than a stack *S*.

5.6 Reflection

Relations. The use of relational rather than functional semantics resulted from the shift to rule rather than structural induction as the basis for the calculation. As we have observed in previous work (Bahr & Hutton, 2015), this shift to rule induction is crucial when the semantics of the source language is no longer compositional, specifically in the case of function application. Therefore, induction on the structure of lambda expressions would not allow us to complete the calculation for the *App* case, because we would lack the required induction hypothesis for the expression x' that originates from a closure.

Partiality. The relational semantics also captures the inherent partiality of the semantics, in the sense that evaluation may fail because of a type mismatch or due to non-termination. As a consequence of this partiality, the specification only explicitly captures one half of compiler correctness, namely completeness, which ensures that compiled code can produce *every* result that is permitted by the semantics. The dual property, soundness, is also important to ensure that compiled code can *only* produce results that are permitted by the

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

semantics. The two example languages we considered earlier both had a total (and deterministic) semantics, so that soundness follows from the specification as well. If we restrict the lambda calculus to a total fragment, such as simply typed lambda terms, soundness also them follows from the specification. In general, however, if our relational semantics is genuinely partial or non-deterministic, we need to explicitly consider both soundness and completeness, as in Hutton & Wright (2007).

Frames. The virtual machine calculated above saves the current memory when a function is applied using the *APP* operation and restores this memory once the function returns using the *RET* operation. An alternative calculation that only saves the registers that are currently being used, which corresponds to the notion of a *stack frame*, is available in the online supplementary material. This alternative version proceeds in the same manner as the above calculation, with the additional administrative overhead of the compiler needing to keep track of the number of registers that are currently being used.

6 Related work

In this section, we summarise some of the main historical developments related to compiler calculation. A more detailed review is provided in Bahr & Hutton (2015).

The idea of formally verifying compilers dates back to the work of McCarthy and Painter (1967) who proved the correctness of a compiler for simple arithmetic expressions. Their handwritten proof was also mechanically checked by Milner and Weyhrauch (1972), establishing the utility of automated proof assistants in the area right from its inception. Since then, a wide range of compiler correctness proofs have been produced, ranging from idealised compilers for small languages up to sophisticated optimisations for real compilers (Dave, 2003). More recently, the CompCert project (Leroy, 2009) demonstrated the feasibility of applying compiler verification in an industrial-strength setting, by mechanically verifying the correctness of an optimising C compiler. This work inspired many further projects on compiler verification, such as Chlipala's verified compiler (Chlipala, 2010), CakeML (Kumar *et al.*, 2014) and DeepSpec (DeepSpec, 2020).

The representation of register memory as a finite partial map from an infinite set of register names to values also appears in the CompCert project. This compiler also uses the pre-order $\sqsubseteq$, but for a different purpose altogether, namely for transformations between two languages that both employ registers. For example, to prove correctness of the various optimisation passes that operate on the intermediate language RTL, CompCert defines corresponding simulation relations. These simulation relations use $\sqsubseteq$ to express the fact that the optimised code uses either more or fewer registers.

Similarly to our compiler, CompCert has to reason about when it is safe to reuse a previously used register. Specifically, this occurs in the compilation pass that translates the Cminor intermediate language, which has an expression fragment, to the RTL intermediate language, which only has simple instructions that address registers. To this end, it maintains a mapping *var* from Cminor variables names to RTL registers and a set *pr* of preserved registers. The correctness property then states that execution of the generated RTL code does not change the contents of the registers appearing in *pr* or in the domain of *var*. By using the pre-order $\sqsubseteq$, we can avoid having to maintain such a set *pr*.

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

Many of the techniques that have traditionally been used to calculate, as opposed to verify, compilers are due to Reynolds (1972). In particular, he introduced three key ideas: definitional interpreters, which express semantics as interpreters written in a compositional manner; continuation-passing style, which allows such interpreters to be rewritten to make control flow explicit; and defunctionalisation, which allows the resulting higher-order interpreters to be rewritten in first-order form. Using these three ideas, Reynolds showed how definitional interpreters could be transformed into abstract machines.

The derivation of compilers from semantics was first considered by Wand (1982a). His approach begins with a continuation semantics, which is then reformulated in point-free form using a generalised composition operator for functions with more than one argument, and custom combinators for manipulating arguments in particular ways. Defunctionalising the resulting semantics then gives rise to a compiler and virtual machine. The original article did not consider the issue of correctness proofs, but in a later article Wand outlined how his compilers could also be proved correct (1982b).

Reynolds and Wand's approaches are both based on writing a semantics in continuation-passing style to capture control flow and then removing the resulting continuations using defunctionalisation. In contrast, our approach fuses these two separate transformations steps into a single calculation step that avoids the need to first introduce and then eliminate continuations (Bahr & Hutton, 2015; Hutton & Bahr, 2016). Our register-based compiler calculations can also be conducted in an unfused manner, but as in our previous stack-based work the fused calculations are simpler, by avoiding the need for continuations, and more direct, by starting from high-level specifications of correctness.

Meijer (1992) developed an algebraic approach to calculating compilers for a variety of languages. His approach starts with a functional semantics expressed as a fold operator (Meijer *et al.*, 1991). He then specifies an equivalent stack-based semantics, for which an implementation is calculated using algebraic properties of folds. Finally, the resulting stack-based machine is defunctionalised to produce a compiler and a virtual machine. While Meijer was able to calculate an impressive range of compilers, starting with a semantics defined as a fold complicates the methodology, and his approach requires significant upfront knowledge about behaviour of the final compiler.

Another approach to deriving compilers was developed by Ager *et al.* (2003b), building on work for abstract machines (Ager *et al.*, 2003a). Their approach begins with a definitional interpreter, from which an abstract machine is derived using Reynold's techniques. This machine is then factorised into a compiler and a virtual machine by introducing a term model that implements a non-standard interpretation of the machine operations. The approach is based on the assumption that all the transformations are semantics preserving, but does not provide a formal proof of the correctness of the resulting compiler.

Finally, for many years, partial evaluation (Jones *et al.*, 1993) has offered the prospect of automatically producing compilers from interpreters. The general technique seeks to apply a program to some of its input to produce a specialised program that, when applied to the rest of its input, gives the same result as the original program, but in a more efficient manner. By writing a partial evaluator in its own subject language it can be applied to itself, which makes it theoretically possible to transform interpreters into compilers (Futamura, 1999). However, there are many challenges that arise in realising this transformation, as explored in a long-standing workshop series (PEPM, 2020).

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

7 Conclusion and further work

In this article, we have shown how our approach to calculating compilers for stack-based machines (Bahr & Hutton, 2015) can be adapted to register-based machines and illustrated its applicability by calculating a series of example compilers. A range of further examples, including features such as local and global state, unbounded loops, lambda calculus with exceptions and exceptions compiled using two code continuations, are available in the online supplementary material (Bahr & Hutton, 2020). In this final section, we reflect on our new approach and discuss directions for further work.

One of the difficulties in calculating compilers for register machines is that in order to make efficient use of memory, it is necessary to reuse registers once their contents are no longer required. For example, given the input $(2 + 3) + 4$, our compiler for arithmetic expressions gives the following code sequence, in which register 0 is first used to store the value 2 and is then reused to store the intermediate result $2 + 3$:

$$\mathit{LOAD}\ 2\ (\mathit{STORE}\ 0\ (\mathit{LOAD}\ 3\ (\mathit{ADD}\ 0\ (\mathit{STORE}\ 0\ (\mathit{LOAD}\ 4\ (\mathit{ADD}\ 0\ \mathit{HALT}))))))$$

In order to achieve this kind of behaviour, the compiler needs to keep track of which registers are not currently 'live' and can hence be safely reused.

This issue was already addressed by McCarthy and Painter in the first published compiler correctness proof (1967). In a similar manner to our first example, the input to their compiler was simple arithmetic expressions, and the output was code for a register machine with an accumulator and an infinite number of additional registers. To tackle the issue of live registers, they used an equivalence relation $=_r$ on memories, defined by:

$$m =_r m' \quad \Leftrightarrow \quad m \text{ and } m' \text{ coincide on registers prior to } r$$

Using our notation, source language and generalisation to code continuations, McCarthy and Painter's compiler correctness property can be written as:

$$\mathit{exec}\ (\mathit{comp}\ e\ r\ c)\ (a, m) = \mathit{exec}\ c\ (\mathit{eval}\ e, m') \quad \text{for some } m' \text{ with } m' =_r m$$

Their proof of this property proceeds by structural induction on the source expression. However, the proof is rather complex and uses a large number of lemmas.

In principle, we could have used the above correctness property as the basis for our calculations. At first sight it appears simpler, because it only involves equality and an equivalence relation. However, it also involves an existentially quantified memory m', with a side condition $m' =_r m$, which makes it difficult and error-prone for use in calculations. In particular, assumptions about such existentially quantified memories have to be remembered across multiple calculation steps and be discharged in the right places. Moreover, we would have to switch to different equivalence relations $=_{r'}$ during the calculation.

Our use of the pre-ordering $\sqsubseteq$ on memories that allows us to 'forget' the contents of registers, together with the monotonicity assumption for *exec*, hides this existential quantification and only requires a single relation $\sqsubseteq$, rather than a family of relations $=_r$. In fact, we can strengthen our generalised compiler correctness property (specification 2) to the following form, which makes the existential quantification explicit:

https://doi.org/10.1017/S0956796820000209 Published online by Cambridge University Press

$$\begin{array}{ll} & \mathit{freeFrom}\ r\ m \\ \Rightarrow & \\ & \mathit{exec}\,(\mathit{comp}\ e\ r\ c)\,(a, m) \;=\; \mathit{exec}\ c\,(\mathit{eval}\ e, m') \quad \text{for some } m' \text{ with } m \sqsubseteq m' \end{array}$$

This strengthened property, with explicit existential quantification, is equivalent to the correctness property in the style of McCarthy and Painter, assuming a partial memory model and monotonicity of *exec*. In contrast, specification 2, which we use for our calculation, is strictly weaker. However, if we consider the top-level correctness property for *compile* rather than *comp*, we can show that specification 1 and the corresponding compiler correctness property in the style of McCarthy and Painter are in fact equivalent.

There are many possible avenues for developing the approach further. Interesting topics for further work include calculating compilers for real-world source languages, such as the core language of the Glasgow Haskell Compiler (GHC); calculating compilers for real-world target machines, such as the Low Level Virtual Machine (LLVM); extending the approach to support typed source and target languages; and developing mechanical tool support to assist with compiler calculations and certify their correctness.

Acknowledgements

Graham Hutton was funded by EPSRC grant EP/P00587X/1, *Mind the Gap: Unified Reasoning About Program Correctness and Efficiency*.

Conflicts of Interest

None.

Supplementary materials

To view supplementary material for this article, please visit https://doi.org/10.1017/S0956796820000209

References

Ager, M. S., Biernacki, D., Danvy, O. & Midtgaard, J. (2003a) A functional correspondence between evaluators and abstract machines. In Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming.

Ager, M. S., Biernacki, D., Danvy, O. & Midtgaard, J. (2003b) *From Interpreter to Compiler and Virtual Machine: A Functional Derivation*. Technical Report RS-03-14. BRICS, Department of Computer Science, University of Aarhus.

Bahr, P. (2014) Proving correctness of compilers using structured graphs. In *Functional and Logic Programming*. LNCS, vol. 8475. Springer.

Bahr, P. & Hutton, G. (2015) Calculating correct compilers. *J. Funct. Program.* **25**.

Bahr, P. & Hutton, G. (2020) *Supplementary Material for Calculating Correct Compilers II*. Available at: https://github.com/pa-ba/reg-machine.

Chaitin, G. (1982) Register allocation & spilling via graph coloring. In Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction.

Chlipala, A. (2010) A verified compiler for an impure functional language. In Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.

Dave, M. (2003) Compiler verification: A bibliography. *Soft. Eng. Notes* **28**(6), 2.
DeepSpec. (2020) *The Science of Deep Specification*. Available at: https://deepspec.org/.
Futamura, Y. (1999) Partial evaluation of computation process – An approach to a compiler-compiler. *Higher-Order Symb. Comput.* **12**(4), 381–391.
Hutton, G. & Bahr, P. (2016) Cutting out continuations. In *A List of Successes That Can Change the World*. Lecture Notes in Computer Science, vol. 9600. Springer.
Hutton, G. & Bahr, P. (2017) Compiling a 50-year journey. *J. Funct. Program.* **27**.
Hutton, G. & Wright, J. (2006) Calculating an exceptional machine. In *Trends in Functional Programming Volume 5*, Loidl, H.-W. (ed). Intellect. Selected papers from the Fifth Symposium on Trends in Functional Programming.
Hutton, G. & Wright, J. (2007) What is the meaning of these constant interruptions? *J. Funct. Program.* **17**(6), 49–64.
Jones, N., Gomard, C. & Sestoft, P. (1993) *Partial Evaluation and Automatic Program Generation*. Prentice-Hall.
Kumar, R., Myreen, M. O., Norrish, M. & Owens, S. (2014) CakeML: A verified implementation of ML. In Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.
Landin, P. J. (1964) The mechanical evaluation of expressions. *Comput. J.* **6**(4), 308–320.
Lattner, C. (2008) Introduction to the LLVM compiler system. In Proceedings of International Workshop on Advanced Computing and Analysis Techniques in Physics Research.
Leroy, X. (2009) Formal verification of a realistic compiler. *Commun. ACM* **52**(7), 107–115.
McCarthy, J. & Painter, J. (1967) Correctness of a compiler for arithmetic expressions. In *Mathematical Aspects of Computer Science*. Proceedings of Symposia in Applied Mathematics, vol. 19. American Mathematical Society.
Meijer, E. (1992) *Calculating Compilers*. PhD Thesis, Katholieke Universiteit Nijmegen.
Meijer, E., Fokkinga, M. & Paterson, R. (1991) Functional programming with bananas, lenses, envelopes and barbed wire. In Functional Programming and Computer Architecture.
Milner, R. & Weyhrauch, R. (1972) Proving compiler correctness in a mechanized logic. *Mach. Intell.* **7**, 51–73.
PEPM. (2020) *Workshop on Partial Evaluation and Program Manipulation*. Available at: https://popl20.sigplan.org/home/pepm-2020.
Reynolds, J. C. (1972) Definitional interpreters for higher-order programming languages. In Proceedings of the ACM Annual Conference.
Spivey, M. (1990) A functional theory of exceptions. *Sci. Comput. Program.* **14**(1), 25–42.
Wadler, P. (1992) Monads for functional programming. In Proceedings of the Marktoberdorf Summer School on Program Design Calculi, Broy, M. (ed). Springer–Verlag.
Wand, M. (1982a) Deriving target code as a representation of continuation semantics. *ACM Trans. Program. Lang. Syst.* **4**(3), 496–517.
Wand, M. (1982b) Semantics-directed machine architecture. In *Principles of Programming Languages*.