Beyond Trees: Calculating Graph-Based Compilers (Functional Pearl)

PATRICK BAHR, IT University of Copenhagen, Denmark
GRAHAM HUTTON, University of Nottingham, United Kingdom

Bahr and Hutton recently developed an approach to compiler calculation that allows a wide range of compilers to be derived from specifications of their correctness. However, a limitation of the approach is that it results in compilers that produce tree-structured code. By contrast, realistic compilers produce code that is essentially graph-structured, where the edges in the graph represent jumps that transfer the flow of control to other locations in the code. In this article, we show how their approach can naturally be adapted to calculate compilers that produce graph-structured code, without changing the underlying calculational methodology, by using a higher-order abstract syntax representation of graphs.

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

Additional Key Words and Phrases: program calculation, graphs, higher-order abstract syntax

ACM Reference Format:
Patrick Bahr and Graham Hutton. 2024. Beyond Trees: Calculating Graph-Based Compilers (Functional Pearl). *Proc. ACM Program. Lang.* 8, ICFP, Article 249 (August 2024), 25 pages. https://doi.org/10.1145/3674638

1 Introduction

The aim of *program calculation* [Backhouse 2003] is to develop programs whose correctness is guaranteed by the manner in which they are constructed. Typically, the process starts with a formal specification for the desired behaviour of a program, from which we then seek to derive an implementation that meets this specification using correctness-preserving reasoning.

In recent work, Bahr and Hutton have developed an approach to calculating *compilers*, which translate high-level programs into lower-level code that machines can execute. Their approach begins with an evaluation function that captures the semantics of the source language. For example, for a simple expression language an evaluation function (*eval*) might map an expression to its value. In contrast, a compilation function (*compile*) maps an expression into code to be executed by a suitable machine (*exec*). Compiler correctness is then expressed by a simple equation,

$$exec \circ compile \;=\; eval$$

which expresses that compilation followed by execution gives the same result as evaluation. Bahr and Hutton's approach shows how to derive the function *compile* by solving such a correctness equation, just like we solve any equation in mathematics. In practice, the equation is usually extended with further concepts such as a stack or an environment, depending on the nature of the source and target languages, but the above captures the basic idea.

Authors' Contact Information: Patrick Bahr, IT University of Copenhagen, Copenhagen, Denmark, paba@itu.dk; Graham Hutton, University of Nottingham, Nottingham, United Kingdom, graham.hutton@nottingham.ac.uk.

ACM 2475-1421/2024/8-ART249
https://doi.org/10.1145/3674638

The methodology initially targeted stack-based machines [Bahr and Hutton 2015], and was used to calculate compilers for a wide range of language features and their combinations, including exceptions, state, lambda calculus, loops, non-determinism and interrupts. The approach was later adapted to register-based machines [Bahr and Hutton 2020], together with well-typed [Pickard and Hutton 2021], non-terminating [Bahr and Hutton 2022] and concurrent [Bahr and Hutton 2023] languages. It also allowed McCarthy and Painter's seminal compiler verification [McCarthy and Painter 1967] to be reworked as a compiler calculation [Hutton and Bahr 2017].

However, a limitation of the above approach is that it results in compilers that produce *tree-structured* code. For example, when compiling a conditional expression **if** *cond* **then** *b1* **else** *b2*, the code for the two branches become parameters for an instruction in the target language. In contrast, a traditional compiler would produce linear code in this case, by concatenating the code for the two branches, and inserting jumps to ensure the appropriate branch is executed. In this manner, we can think of traditional compilers as producing *graph-structured* code, where the edges in the graph represent jumps that transfer the flow of control. Viewing code as graphs also allows a number of other control flow concepts to be naturally considered, such as sharing (multiple jumps to a single piece of code), and looping (backward jumps to repeat a piece of code.)

Unfortunately, graphs and program calculation are not usually regarded as being natural bedfellows. In particular, programming with graphs often requires the generation and management of some form of labels or names that are used to represent the edges in graphs, which can significantly complicate the progress of reasoning about the resulting programs. For example, the verification of a compiler for a simple language with exceptions in [Hutton and Wright 2004] requires a considerable number of lemmas regarding how jump labels are managed.

In this article, we avoid such issues by using a representation of graphs that does not require explicit labels. In particular, we utilise the approach of Oliveira and Cook [2012], in which graphs are represented using *parametric higher-order abstract syntax* (PHOAS) [Chlipala 2008]. Using this approach, we can leverage the binding and naming mechanisms of the host language to avoid having to explicitly deal with these concepts. More technically, in this article we:

- Show how Bahr and Hutton's compiler calculation methodology can be naturally adapted to produce graph-structured code. Using PHOAS as the basis to calculate a graph-based definition from a tree-based definition is the main technical idea that enables this.
- Show how compilers for loops can be calculated using a coinductively-defined language of infinite trees, which is then refined into a graph-based target language via calculation. The key idea in such calculations is the use of infinite trees as an intermediate structure.
- Introduce our approach using three examples of increasing complexity, starting with conditionals (Sections 2 and 3), then looping (Section 4), and finally state (Section 5).

Our approach is generally applicable and can be combined with other methods that derive tree-based compilers, such as Meijer [1992], Wand [1995], and Ager et al. [2003]. It also removes a restriction of previous work, by allowing the calculation of compilers that produce cyclic code, which cannot be represented with finite tree-structured code. In this manner, the graph-based approach that is presented does not invalidate previous work that produces tree-structured code, but rather complements it and demonstrates its relevance for practical compilers.

All our programs and calculations are written in Haskell notation, but for reasoning purposes we assume the language is total. Whereas in many articles calculations are often omitted or compressed for brevity, here they are the central focus so are typically presented in detail. All the calculations have been formalised in the Agda proof assistant [Norell 2007], and are available online

as supplementary material [Bahr and Hutton 2024]. The formalisation also includes a number of additional examples, including exceptions and the lambda calculus.

2 Calculating a Tree-Based Compiler

To illustrate the issue with Bahr and Hutton's compiler calculation methodology, we begin in this section by calculating a compiler for a simple expression language comprising integers, addition and conditionals, whose syntax and semantics is defined as follows:

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

$$\begin{aligned}
&\mathit{eval} :: \mathit{Expr} \to \mathit{Int}\\
&\mathit{eval}\ (\mathit{Val}\ n) \quad\ = n\\
&\mathit{eval}\ (\mathit{Add}\ x\ y) = \mathit{eval}\ x + \mathit{eval}\ y\\
&\mathit{eval}\ (\mathit{If}\ x\ y\ z) = \textbf{if}\ \mathit{eval}\ x == 0\ \textbf{then}\ \mathit{eval}\ z\ \textbf{else}\ \mathit{eval}\ y
\end{aligned}$$

For the purpose of conditionals, we follow the standard C-like convention where the logical value false is represented by the integer zero, and true is represented by any non-zero integer.

2.1 Compiler Specification

Given the above definitions, we now aim to calculate a compilation function $\mathit{compile} :: \mathit{Expr} \to \mathit{Code}$ that translates expressions of the source language into code for our target language. We assume that the target language is a stack machine, whose semantics is given by a function $\mathit{exec} :: \mathit{Code} \to \mathit{Stack} \to \mathit{Stack}$, where stacks are simply represented as lists of integers:

$$\textbf{type}\ \mathit{Stack} = [\mathit{Int}]$$

Neither the target language *Code* nor its semantics *exec* are given up front, but will instead fall out of the calculation of the compiler. Prior to specifying the desired behaviour of the compiler, we generalise the function *compile* to a function $\mathit{comp} :: \mathit{Expr} \to \mathit{Code} \to \mathit{Code}$ that takes an additional code argument to be executed after the compiled code. The addition of such a *code continuation* is a key aspect of the methodology [Bahr and Hutton 2015] and simplifies the resulting calculations. Correctness of the generalised compiler can then be expressed by the following equation:

$$\mathit{exec}\ c\ (\mathit{eval}\ x : s) \quad = \quad \mathit{exec}\ (\mathit{comp}\ x\ c)\ s \tag{1}$$

That is, executing the result of compiling an expression followed by additional code gives the same result as executing the additional code with the value of the expression on top of the stack.

2.2 Compiler Calculation

We now calculate the compiler by proving equation (1) by induction on the expression x. For each case, we aim to transform the left side of the equation, $\mathit{exec}\ c\ (\mathit{eval}\ x : s)$, into the form $\mathit{exec}\ c'\ s$ for some code c'. We can then define $\mathit{comp}\ x\ c = c'$ for that case, which by construction satisfies the correctness equation. The base case, $x = \mathit{Val}\ n$, proceeds as follows:

$$\begin{aligned}
&\quad \mathit{exec}\ c\ (\mathit{eval}\ (\mathit{Val}\ n) : s)\\
&= \quad \{\text{apply } \mathit{eval} \text{ for } \mathit{Val}\}\\
&\quad \mathit{exec}\ c\ (n : s)\\
&= \quad \{\text{define: } \mathit{exec}\ (\mathit{PUSH}\ n\ c)\ s = \mathit{exec}\ c\ (n : s)\}\\
&\quad \mathit{exec}\ (\mathit{PUSH}\ n\ c)\ s
\end{aligned}$$

To obtain a term of the required form, in the final step above we introduced a new constructor $\mathit{PUSH} :: \mathit{Int} \to \mathit{Code} \to \mathit{Code}$ for the *Code* type, along with a clause of the *exec* function, which

defines the semantics of *PUSH*. The final term of the calculation is then of the form $exec\ c'\ s$ with $c' = PUSH\ n\ c$, which gives the first clause for the definition of the compiler:

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

The inductive case for addition proceeds in a similar manner, in which we introduce a new code constructor *ADD* that adds the top two values on the stack, this time motivated by the desire to transform the term into a form to which the induction hypotheses can be applied:

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

The final term is now of the required form, which gives the next clause for the compiler:

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

Finally, the case for conditionals proceeds as follows, in which we use the fact that in a total language we can freely distribute functions over conditional expressions:

$$\begin{array}{ll}
 & exec\ c\ (eval\ (If\ x\ y\ z) : s) \\
= & \{\text{ apply } eval \text{ for } If\ \} \\
 & exec\ c\ ((\textbf{if}\ eval\ x == 0\ \textbf{then}\ eval\ z\ \textbf{else}\ eval\ y) : s) \\
= & \{\text{ distribute } (:s) \text{ over conditional }\} \\
 & exec\ c\ (\textbf{if}\ eval\ x == 0\ \textbf{then}\ eval\ z : s\ \textbf{else}\ eval\ y : s) \\
= & \{\text{ distribute } exec\ c \text{ over conditional }\} \\
 & \textbf{if}\ eval\ x == 0\ \textbf{then}\ exec\ c\ (eval\ z : s)\ \textbf{else}\ exec\ c\ (eval\ y : s) \\
= & \{\text{ induction hypotheses for } y \text{ and } z\ \} \\
 & \textbf{if}\ eval\ x == 0\ \textbf{then}\ exec\ (comp\ z\ c)\ s\ \textbf{else}\ exec\ (comp\ y\ c)\ s \\
= & \{\text{ define: } exec\ (JPZ\ c'\ c)\ (n : s) = \textbf{if}\ n == 0\ \textbf{then}\ exec\ c'\ s\ \textbf{else}\ exec\ c\ s\ \} \\
 & exec\ (JPZ\ (comp\ z\ c)\ (comp\ y\ c))\ (eval\ x : s) \\
= & \{\text{ induction hypothesis for } x\ \} \\
 & exec\ (comp\ x\ (JPZ\ (comp\ z\ c)\ (comp\ y\ c)))\ s
\end{array}$$

To apply the induction hypothesis for the expression x, in the penultimate step above we introduced a new code constructor *JPZ* that jumps to alternative code if the top of the stack is zero. The final term of the calculation then gives the final clause for the compiler:

$$comp\ (If\ x\ y\ z)\ c = comp\ x\ (JPZ\ (comp\ z\ c)\ (comp\ y\ c))$$

We conclude this section by returning to the top-level compilation function $compile :: Expr \to Code$, whose correctness can be expressed by the following equation:

$$eval\ x : s \quad = \quad exec\ (compile\ x)\ s \tag{2}$$

In a similar manner to equation (1) for *comp*, we aim to transform the left side of this equation, $eval\ x : s$, into the form $exec\ c\ s$ for some code c, so that we can then define $compile\ x = c$. In this

$$
\begin{array}{l}
\textbf{data}\ \mathit{Code} = \mathit{HALT} \mid \mathit{PUSH}\ \mathit{Int}\ \mathit{Code} \mid \mathit{ADD}\ \mathit{Code} \mid \mathcal{JPZ}\ \mathit{Code}\ \mathit{Code} \\
\mathit{compile} :: \mathit{Expr} \to \mathit{Code} \\
\mathit{compile}\ x = \mathit{comp}\ x\ \mathit{HALT} \\
\mathit{comp} :: \mathit{Expr} \to \mathit{Code} \to \mathit{Code} \\
\begin{array}{lll}
\mathit{comp}\ (\mathit{Val}\ n) & c & = \mathit{PUSH}\ n\ c \\
\mathit{comp}\ (\mathit{Add}\ x\ y) & c & = \mathit{comp}\ x\ (\mathit{comp}\ y\ (\mathit{ADD}\ c)) \\
\mathit{comp}\ (\mathit{If}\ x\ y\ z) & c & = \mathit{comp}\ x\ (\mathcal{JPZ}\ (\mathit{comp}\ z\ c)\ (\mathit{comp}\ y\ c))
\end{array} \\
\mathit{exec} :: \mathit{Code} \to \mathit{Stack} \to \mathit{Stack} \\
\begin{array}{lll}
\mathit{exec}\ \mathit{HALT} & s & = s \\
\mathit{exec}\ (\mathit{PUSH}\ n\ c) & s & = \mathit{exec}\ c\ (n : s) \\
\mathit{exec}\ (\mathit{ADD}\ c) & (n : m : s) & = \mathit{exec}\ c\ (m + n : s) \\
\mathit{exec}\ (\mathcal{JPZ}\ c'\ c) & (n : s) & = \textbf{if}\ n == 0\ \textbf{then}\ \mathit{exec}\ c'\ s\ \textbf{else}\ \mathit{exec}\ c\ s \\
\mathit{exec}\ \_ & \_ & = [\,]
\end{array}
\end{array}
$$

Fig. 1. Tree-based compiler for conditional expressions.

case we don't need induction, as introducing a new code constructor *HALT* that returns the current stack allows the correctness equation for *comp* to be used to obtain the required form:

$$
\begin{array}{ll}
 & \mathit{eval}\ x : s \\
= & \quad \{\, \text{define: } \mathit{exec}\ \mathit{HALT}\ s = s \,\} \\
 & \mathit{exec}\ \mathit{HALT}\ (\mathit{eval}\ x : s) \\
= & \quad \{\, \text{correctness of } \mathit{comp} - (1) \,\} \\
 & \mathit{exec}\ (\mathit{comp}\ x\ \mathit{HALT})\ s
\end{array}
$$

In summary, we have calculated the definitions shown in Figure 1. The definition of *exec* also includes the catch-all clause $\mathit{exec}\ \_\ \_ = [\,]$, which we have added to make the definition total. The choice for this catch-all clause is not important as it plays no role in the calculation.

3 Calculating a Graph-Based Compiler

Two characteristics of the code produced by the compiler calculated in the previous section make it unrealistic in practice. First of all, code for a realistic machine is linear, i.e. each instruction is followed by at most one other instruction to be executed next. This is true for the *HALT*, *PUSH* and *ADD* instructions, but not the $\mathcal{JPZ}$ instruction, which is followed by two possible pieces of code to be executed next, one for each branch of a conditional expression:

$$\mathcal{JPZ} :: \mathit{Code} \to \mathit{Code} \to \mathit{Code}$$

In this manner, the compiler produces *tree-structured* code. In a realistic instruction set, we would expect the $\mathcal{JPZ}$ instruction to have type $\mathit{Loc} \to \mathit{Code} \to \mathit{Code}$, i.e. the first argument is not code itself but rather a location that points to a piece of code, to ensure that code is linear.

Secondly, when compiling a conditional expression, the compilation function *comp* duplicates code by passing the additional code c to two recursive calls of the compiler:

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

In a realistic compiler, we would instead expect to find an instruction $\mathcal{JMP} :: \mathit{Loc} \to \mathit{Code}$ that allows us to jump to the same piece of code multiple times, rather than duplicating code.

3.1 Graph-Structured Code

In light of the above observations, we could define a graph-based version of the code type,

$$\textbf{data}\ \mathit{Code}_g = \mathit{HALT}_g \mid \mathit{PUSH}_g\ \mathit{Int}\ \mathit{Code}_g \mid \mathit{ADD}_g\ \mathit{Code}_g \mid \mathit{JPZ}_g\ \mathit{Loc}\ \mathit{Code}_g \mid \mathit{JMP}_g\ \mathit{Loc}$$

where jump locations are simply integer addresses within the code, i.e. the first instruction is located at address zero, the second at address one, and so on:

$$\textbf{type}\ \mathit{Loc} = \mathit{Int}$$

However, the use of explicit jump locations complicates the compiler implementation, and makes reasoning about it difficult. Address calculation is delicate on its own, and it would be beneficial to separate this from the process of compiler calculation. To this end, we treat jump locations *abstractly* by using *parametric higher-order abstract syntax* [Chlipala 2008] to provide symbolic locations, which we call *labels*. We achieve this by abstracting the *Loc* type to a type variable l, and adding a new instruction $\mathit{LAB}_g^{\rightarrow}$ that brings a fresh label into scope using a function parameter:

$$\begin{aligned}\textbf{data}\ \mathit{Code}_g\ l = {} & \mathit{HALT}_g \mid \mathit{PUSH}_g\ \mathit{Int}\ (\mathit{Code}_g\ l) \mid \mathit{ADD}_g\ (\mathit{Code}_g\ l) \mid \mathit{JPZ}_g\ l\ (\mathit{Code}_g\ l) \mid \mathit{JMP}_g\ l \mid \\ & \mathit{LAB}_g^{\rightarrow}\ (l \rightarrow \mathit{Code}_g\ l)\ (\mathit{Code}_g\ l)\end{aligned}$$

Intuitively, $\mathit{JMP}_g\ l$ jumps to the code with label l, while $\mathit{LAB}_g^{\rightarrow}\ (\lambda l \rightarrow c)\ c'$ labels the code c' with a fresh label l and makes that label available in the code c so that it can jump to c'. For example, the following piece of linear code, written in an assembly-like notation,

```
      PUSH 2
      JPZ  l
      PUSH 3
      JMP  l
  l:  PUSH 4
      HALT
```

can be represented by the following term of type $\mathit{Code}_g\ l$:

$$\mathit{LAB}_g^{\rightarrow}\ (\lambda l \rightarrow \mathit{PUSH}_g\ 2\ (\mathit{JPZ}_g\ l\ (\mathit{PUSH}_g\ 3\ (\mathit{JMP}_g\ l)))) \ (\mathit{PUSH}_g\ 4\ \mathit{HALT}_g)$$

Graph-based code represented using the type $\mathit{Code}_g\ l$ has the desired linear structure. While the constructor $\mathit{LAB}_g^{\rightarrow}$ may appear non-linear since it essentially takes two code arguments, code of the form $\mathit{LAB}_g^{\rightarrow}\ (\lambda l \rightarrow c)\ c'$ represents the *sequential composition* of two pieces of code as shown in the example above. Note that only forward jumps are possible at present, because in $\mathit{LAB}_g^{\rightarrow}\ (\lambda l \rightarrow c)\ c'$ the label l is only in scope in the first piece of code c, allowing jumps to be made to the second piece of code c'. The forward arrow in $\mathit{LAB}_g^{\rightarrow}$ reflects this fact. We will introduce an instruction $\mathit{LAB}_g^{\leftarrow}$ that permits backward jumps in Section 4 when we consider a language with looping.

The type Code_g is an instance of the notion of a *structured graph* [Oliveira and Cook 2012]. Each structured graph type comes equipped with a canonical function $(\!|\cdot|\!)$ that unravels a graph to its corresponding tree, which in our case is defined as follows:

$$\begin{aligned}
&(\!|\cdot|\!) :: \mathit{Code}_g\ \mathit{Code} \rightarrow \mathit{Code} \\
&(\!|\mathit{HALT}_g|\!) \quad\ = \mathit{HALT} \\
&(\!|\mathit{PUSH}_g\ n\ c|\!) = \mathit{PUSH}\ n\ (\!|c|\!) \\
&(\!|\mathit{ADD}_g\ c|\!) \quad\ = \mathit{ADD}\ (\!|c|\!) \\
&(\!|\mathit{JPZ}_g\ l\ c|\!) \quad = \mathit{JPZ}\ l\ (\!|c|\!) \\
&(\!|\mathit{JMP}_g\ l|\!) \quad\ = l \\
&(\!|\mathit{LAB}_g^{\rightarrow}\ f\ c|\!) \ = (\!|f\ (\!|c|\!)|\!)
\end{aligned}$$

That is, the function $(\!|\cdot|\!)$ takes a graph of type $Code_g\ Code$ in which the free occurrences of abstract labels have already been replaced by concrete trees of type $Code$, and converts the graph into a tree. Note that in the case of $\mathcal{J}MP_g\ l$ we simply return the code l that is supplied as a parameter, while for $LAB_g^{\rightarrow}\ f\ c$ we first unravel the graph c, then apply the function f to the resulting code, and finally unravel the resulting graph. For instance, applying $(\!|\cdot|\!)$ to the example code graph above gives the following code tree, in which the code $PUSH\ 4\ HALT$ at label l occurs twice:

$PUSH\ 2\ (\mathcal{J}PZ\ (PUSH\ 4\ HALT)\ (PUSH\ 3\ (PUSH\ 4\ HALT)))$

Structured graphs rely on parametricity in the label type l to ensure that labels can only be treated purely symbolically, i.e. they cannot be inspected. In our case, this means that we must only construct graphs of the polymorphic type $\forall\ l.\ Code_g\ l$. The unravelling function $(\!|\cdot|\!)$ then instantiates this polymorphic type to $Code_g\ Code$, so that in the case of $(\!|LAB_g^{\rightarrow}\ (\lambda l \rightarrow c')\ c|\!)$ it can use function application to substitute the free occurrences of label l in c' with the unravelled code $(\!|c|\!)$. Parametricity ensures that the labels provided by $LAB_g^{\rightarrow}$ can only be used as names [Atkey 2009], but otherwise doesn't play a role in our compiler calculations.

3.2 Compiler Specification

Given the above definitions, our aim now is to calculate a compiler that produces graph-structured code of type $\forall\, l.\ Code_g\ l$. First of all, the semantics $exec_g$ of such code can be defined by:

$$
\begin{aligned}
&exec_g :: (\forall\, l.\ Code_g\ l) \rightarrow Stack \rightarrow Stack \\
&exec_g\ c\ s = exec\ (\!|c|\!)\ s
\end{aligned}
$$

In this manner, the execution of graph-structured code comprises two separate parts: the first part, given by $(\!|\cdot|\!)$, specifies how to execute jump instructions, while the second part, given by $exec$, specifies how to execute all other instructions. Note that the parametric type $\forall\, l.\ Code_g\ l$ of the argument c is implicitly instantiated to $Code_g\ Code$ before it is passed to $(\!|\cdot|\!)$.

The correctness of a graph-based compilation function $comp_g :: Expr \rightarrow Code_g\ l \rightarrow Code_g\ l$ can now be expressed by the following equation, which has the same form as equation (1) for $comp$ in Section 2, except that it operates on code graphs rather than code trees:

$$exec_g\ c\ (eval\ x : s) \quad = \quad exec_g\ (comp_g\ x\ c)\ s \tag{3}$$

The above equation can then be simplified as follows:

$$
\begin{aligned}
& exec_g\ c\ (eval\ x : s) = exec_g\ (comp_g\ x\ c)\ s \\
\Leftrightarrow \quad & \{\text{ apply } exec_g \text{ }\} \\
& exec\ (\!|c|\!)\ (eval\ x : s) = exec\ (\!|comp_g\ x\ c|\!)\ s \\
\Leftrightarrow \quad & \{\text{ correctness of } comp \text{ }\} \\
& exec\ (comp\ x\ (\!|c|\!))\ s = exec\ (\!|comp_g\ x\ c|\!)\ s \\
\Leftarrow \quad & \{\text{ congruence }\} \\
& comp\ x\ (\!|c|\!) = (\!|comp_g\ x\ c|\!)
\end{aligned}
$$

In conclusion, to establish the correctness of the compilation function $comp_g$ it suffices to satisfy the following simple equation, which provides a suitable specification from which we can calculate the graph-based compiler $comp_g$ from the tree-based compiler $comp$:

$$comp\ x\ (\!|c|\!) \quad = \quad (\!|comp_g\ x\ c|\!) \tag{4}$$

3.3 Compiler Calculation

In a similar manner to Section 2, the calculation proceeds from equation (4) by induction on the expression x. For each case, we aim to transform the left side of the equation, $comp\ x\ (\!|c|\!)$, into the form $(\!|c'|\!)$ for some code c'. We can then define $comp_g\ x\ c = c'$ for that case, which by construction satisfies the correctness equation. Unlike the calculation in Section 2, the function $comp_g$ is the only unknown in equation (4), so no further definitions need to be discovered during the calculation. The cases for values and addition are straightforward:

$$
\begin{array}{ll}
 & comp\ (Val\ n)\ (\!|c|\!) \\
= & \quad \{\text{ apply } comp \text{ for } Val \} \\
 & PUSH\ n\ (\!|c|\!) \\
= & \quad \{\text{ unapply } (\!|\cdot|\!) \text{ for } PUSH_g \} \\
 & (\!|PUSH_g\ n\ c|\!)
\end{array}
$$

and

$$
\begin{array}{ll}
 & comp\ (Add\ x\ y)\ (\!|c|\!) \\
= & \quad \{\text{ apply } comp \text{ for } Add \} \\
 & comp\ x\ (comp\ y\ (ADD\ (\!|c|\!))) \\
= & \quad \{\text{ unapply } (\!|\cdot|\!) \text{ for } ADD_g \} \\
 & comp\ x\ (comp\ y\ (\!|ADD_g\ c|\!)) \\
= & \quad \{\text{ induction hypothesis for } y \} \\
 & comp\ x\ (\!|comp_g\ y\ (ADD_g\ c)|\!) \\
= & \quad \{\text{ induction hypothesis for } x \} \\
 & (\!|comp_g\ x\ (comp_g\ y\ (ADD_g\ c))|\!)
\end{array}
$$

The case for conditionals is more involved, as this is where the move to graph-based code becomes important. The calculation begins by applying *comp*, abstracting over the resulting duplicated term $(\!|c|\!)$ to avoid code duplication, and then using the definition of $(\!|\cdot|\!)$ to transform the term into a form to which the induction hypothesis for the two branches y and z can be applied:

$$
\begin{array}{ll}
 & comp\ (If\ x\ y\ z)\ (\!|c|\!) \\
= & \quad \{\text{ apply } comp \text{ for } If \} \\
 & comp\ x\ (\mathcal{J}PZ\ (comp\ z\ (\!|c|\!))\ (comp\ y\ (\!|c|\!))) \\
= & \quad \{\text{ abstract over } (\!|c|\!) \} \\
 & comp\ x\ ((\lambda l \to \mathcal{J}PZ\ (comp\ z\ l)\ (comp\ y\ l))\ (\!|c|\!)) \\
= & \quad \{\text{ unapply } (\!|\cdot|\!) \text{ for } \mathcal{J}MP_g \} \\
 & comp\ x\ ((\lambda l \to \mathcal{J}PZ\ (comp\ z\ (\!|\mathcal{J}MP_g\ l|\!))\ (comp\ y\ (\!|\mathcal{J}MP_g\ l|\!)))\ (\!|c|\!)) \\
= & \quad \{\text{ induction hypothesis for } y \text{ and } z \} \\
 & comp\ x\ ((\lambda l \to \mathcal{J}PZ\ (\!|comp_g\ z\ (\mathcal{J}MP_g\ l)|\!)\ (\!|comp_g\ y\ (\mathcal{J}MP_g\ l)|\!))\ (\!|c|\!))
\end{array}
$$

We then proceed by abstracting over the first argument of the $\mathcal{J}PZ$ instruction, which is motivated by the desire for this argument to become an abstract jump label, and then transform the term into a form to which the induction hypothesis for x can be applied:

$$
\begin{array}{ll}
 & comp\ x\ ((\lambda l \to \mathcal{J}PZ\ (\!|comp_g\ z\ (\mathcal{J}MP_g\ l)|\!)\ (\!|comp_g\ y\ (\mathcal{J}MP_g\ l)|\!))\ (\!|c|\!)) \\
= & \quad \{\text{ abstract over } (\!|comp_g\ z\ (\mathcal{J}MP_g\ l)|\!) \} \\
 & comp\ x\ ((\lambda l \to (\lambda l' \to \mathcal{J}PZ\ l'\ (\!|comp_g\ y\ (\mathcal{J}MP_g\ l)|\!))\ (\!|comp_g\ z\ (\mathcal{J}MP_g\ l)|\!))\ (\!|c|\!)) \\
= & \quad \{\text{ unapply } (\!|\cdot|\!) \text{ for } \mathcal{J}PZ_g \} \\
 & comp\ x\ ((\lambda l \to (\lambda l' \to (\!|\mathcal{J}PZ_g\ l'\ (comp_g\ y\ (\mathcal{J}MP_g\ l))|\!))\ (\!|comp_g\ z\ (\mathcal{J}MP_g\ l)|\!))\ (\!|c|\!))
\end{array}
$$

$$
\begin{aligned}
&\mathit{compile}_g :: \mathit{Expr} \to \mathit{Code}_g\ l \\
&\mathit{compile}_g\ x = \mathit{comp}_g\ x\ \mathit{HALT}_g \\
&\mathit{comp}_g :: \mathit{Expr} \to \mathit{Code}_g\ l \to \mathit{Code}_g\ l \\
&\mathit{comp}_g\ (\mathit{Val}\ n)\quad\ \ c = \mathit{PUSH}_g\ n\ c \\
&\mathit{comp}_g\ (\mathit{Add}\ x\ y)\ c = \mathit{comp}_g\ x\ (\mathit{comp}_g\ y\ (\mathit{ADD}_g\ c)) \\
&\mathit{comp}_g\ (\mathit{If}\ x\ y\ z)\ c = \mathit{comp}_g\ x\ (\mathit{LAB}_g^{\to}\ (\lambda l \to \mathit{LAB}_g^{\to}\ (\lambda l' \to \\
&\qquad\qquad\qquad\qquad \mathit{JPZ}_g\ l'\ (\mathit{comp}_g\ y\ (\mathit{JMP}_g\ l)))\ (\mathit{comp}_g\ z\ (\mathit{JMP}_g\ l)))\ c)
\end{aligned}
$$

Fig. 2. Graph-based compiler for conditional expressions.

$$
\begin{aligned}
&= \quad \{\text{ unapply } (\!|\cdot|\!) \text{ for } \mathit{LAB}_g^{\to}\ \} \\
&\quad \mathit{comp}\ x\ ((\lambda l \to (\!|\mathit{LAB}_g^{\to}\ (\lambda l' \to \mathit{JPZ}_g\ l'\ (\mathit{comp}_g\ y\ (\mathit{JMP}_g\ l)))\ (\mathit{comp}_g\ z\ (\mathit{JMP}_g\ l))|\!))\ (\!|c|\!)) \\
&= \quad \{\text{ unapply } (\!|\cdot|\!) \text{ for } \mathit{LAB}_g^{\to}\ \} \\
&\quad \mathit{comp}\ x\ (\!|\mathit{LAB}_g^{\to}\ (\lambda l \to \mathit{LAB}_g^{\to}\ (\lambda l' \to \mathit{JPZ}_g\ l'\ (\mathit{comp}_g\ y\ (\mathit{JMP}_g\ l)))\ (\mathit{comp}_g\ z\ (\mathit{JMP}_g\ l)))\ c|\!) \\
&= \quad \{\text{ induction hypothesis for } x\ \} \\
&\quad (\!|\mathit{comp}_g\ x\ (\mathit{LAB}_g^{\to}\ (\lambda l \to \mathit{LAB}_g^{\to}\ (\lambda l' \to \mathit{JPZ}_g\ l'\ (\mathit{comp}_g\ y\ (\mathit{JMP}_g\ l)))\ (\mathit{comp}_g\ z\ (\mathit{JMP}_g\ l)))\ c)|\!)
\end{aligned}
$$

The final term above now has the required form, which completes the calculation for this case. Note that in the two steps before the final step, we use the fact that the unravelling function $(\!|\cdot|\!)$ for the case of labels can be beta-expanded to $(\!|\mathit{LAB}_g^{\to}\ f\ c|\!) = (\lambda l \to (\!|f\ l|\!))\ (\!|c|\!)$.

The top-level compilation function $\mathit{compile}_g :: \mathit{Expr} \to \mathit{Code}_g\ l$ can be calculated in a similar manner from the tree-based version $\mathit{compile} :: \mathit{Expr} \to \mathit{Code}$ using the specification $\mathit{compile}\ x = (\!|\mathit{compile}_g\ x|\!)$. Figure 2 summarises all calculated definitions. In conclusion, we have derived a graph-based compiler that uses explicit labels and jumps to avoid the two issues with the previous version, namely tree-structured code and code duplication. For example, applying $\mathit{compile}_g$ to the expression $\mathit{If}\ (\mathit{Val}\ 2)\ (\mathit{Val}\ 3)\ (\mathit{Val}\ 4)$ gives the code graph

$$\mathit{PUSH}_g\ 2\ (\mathit{LAB}_g^{\to}\ (\lambda l \to \mathit{LAB}_g^{\to}\ (\lambda l' \to \mathit{JPZ}_g\ l'\ (\mathit{PUSH}_g\ 3\ (\mathit{JMP}_g\ l)))\ (\mathit{PUSH}_g\ 4\ (\mathit{JMP}_g\ l)))\ \mathit{HALT}_g)$$

which corresponds to the following code in assembly-like notation:

```
      PUSH 2
      JPZ  l'
      PUSH 3
      JMP  l
l' :  PUSH 4
      JMP  l
l :   HALT
```

This code is both linear, i.e. each instruction is followed by at most one instruction, and avoids duplication, i.e. there are two jumps to the code at label l rather than this code being duplicated. For this simple example the code at label l is just HALT, but in general it could be an arbitrarily large piece of code, which would be duplicated in the original tree-based function *compile*.

3.4 Reflection

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

Explicit versus generic definitions. The first step in calculating a graph-based compiler is the definition of the type Code_g of code graphs, and the function $(\!|\cdot|\!)$ that unravels a code graph to

the corresponding code tree. Both follow in a generic manner from the type *Code* of code trees. This can be made explicit by defining a generic tree type *Tree i* and a corresponding graph type *Graph i l*, parameterised by a type *i* of instructions. For example, given the instruction type

data $Inst\ l = PUSH\ Int \mid ADD \mid \mathcal{J}PZ\ l$

we can show that *Code* is isomorphic to *Tree Inst* and that $Code_g\ l$ is isomorphic to *Graph Inst l*. The unravelling function $(\!|\cdot|\!)$ can be defined as a generic function of type $\forall\, l.\ Graph\ i\ l \to Tree\ i$, independent of the instruction type *i*. In turn, we can implement a generic function $seqn :: \forall\, l.\ Graph\ i\ l \to Seqn\ i$ that turns a graph into a sequence of instructions with explicit code pointers. By composing a graph-based compiler with the generic function *seqn*, we can obtain a compiler that produces linear code with explicit jumps. Details can be found in the accompanying Agda code. For simplicity of exposition, in this article we follow the approach of [Bahr and Hutton 2015] and use explicit definitions for the basic types and functions, rather than generic definitions.

Two-step calculation. The methodology [Bahr and Hutton 2015] that we used to calculate the tree-based compiler *comp* can be viewed as a fusion of three separate calculation steps: introducing a stack, introducing continuations, and defunctionalisation. It is thus natural to ask whether the calculation of the graph-based compiler $comp_g$ from the tree-based version *comp* can be fused with the calculation of *comp* from the source language semantics *eval*, such that we can calculate $comp_g$ directly from *eval*. However, our calculation technique for producing graph-based compilers depends crucially on a two-step process, which prevents us from fusing the two steps together. We can see this by considering the correctness specification for $comp_g$:

$$exec_g\ c\ (eval\ x : s) \quad = \quad exec_g\ (comp_g\ x\ c)\ s$$

On its own, this equation does not express the desire for $comp_g$ to produce graph-structured code. Indeed, the above equation is equivalent to the correctness specification for *comp*,

$$exec\ c\ (eval\ x : s) \quad = \quad exec\ (comp\ x\ c)\ s$$

if the type $Code_g$ and function $exec_g$ are considered as unknowns, in the same way that *Code* and *exec* are unknowns for the latter equation. Instead, the desire for $comp_g$ to produced graph-shaped code is expressed explicitly in the definition of $Code_g$ and $exec_g$, which are given prior to calculation and are thus part of the specification. Moreover, as we have seen, these definitions follow in a systematic manner from the calculation of the tree-based compiler *comp*.

Branching control flow. In essence, the method that we presented here allows us to calculate compilers that produce code with branching control flow. This branching control flow manifests itself as the tree-structured code produced by the original compiler *comp*. However, the tree-structure itself is not necessarily the issue, but rather the underlying branching control flow. The syntax of the source language is also tree-structured, but its semantics is not necessarily branching. In particular, the semantics of addition in the source language is not branching and indeed *comp* produces linear code for it. Wand [1982] and Gibbons [2021] have used this insight to derive a compiler for a simple expression language without branching control flow by linearising the tree structure of expressions using the associativity of function composition. However, this linearisation crucially depends on the fact that the control flow of the language is not branching.

Parametricity. The use of parametric higher-order abstract syntax avoids having to explicitly deal with names and requires little additional technical machinery. A drawback is that reasoning about parametricity itself is underdeveloped, and often not possible in proof assistants. For example, such reasoning is necessary if we wished to calculate a direct implementation of the graph-based

machine $exec_g$ by fusing together the *exec* and $(\!|\cdot|\!)$ functions in the definition $exec_g\ c\ s = exec\ (\!|c|\!)\ s$. In the accompanying Agda formalisation, we give an example of such a calculation.

4 Compiling to Cyclic Code

Code produced by the compiler in the previous section can jump *forward* to other code locations. However, for source languages that feature cyclic control flow, e.g. in the form of loops, we also need to produce code that can jump *backward*. For example, the following piece of code in an assembly-like notation repeatedly adds two to the top of the stack:

```
     PUSH 0
l :  PUSH 2
     ADD
     JMP l
```

To illustrate such looping, we consider a simple expression language with an infinite repetition primitive, whose syntax and semantics can be defined as follows:

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

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

This language provides a degenerate form of looping, but suffices to illustrate the core reasoning principles prior to considering a more realistic language that features while loops in Section 5. However, there is a problem with the above definition for the function *eval*: we assume that our meta-language is total, but *eval* (*Repeat x*) does not terminate and hence *eval* is partial.

To express non-terminating computations, we require a means to explicitly represent them. The *partiality monad* [Capretta 2005; Danielsson 2012] provides this functionality:

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

Intuitively, *Now v* represents a computation that immediately returns the value *v*, while *Later p* delays the computation *p* by one time step. The partiality type is defined coinductively, which we indicate above by writing **codata**. In particular, this means that we can delay a computation infinitely often to represent a non-terminating computation, such as:

$$\begin{aligned}
&never :: Partial\ a\\
&never = Later\ never
\end{aligned}$$

While the definition for *never* is non-terminating, it is well-defined as the recursive call is guarded by the coinductive constructor *Later*. The partiality type forms a monad under the following definitions, which allows the **do** notation to be used with this type:

$$\begin{aligned}
&return :: a \to Partial\ a\\
&return = Now
\end{aligned}$$

$$\begin{aligned}
&(\gg\!=) :: Partial\ a \to (a \to Partial\ b) \to Partial\ b\\
&(Now\ x)\ \ \gg\!= f = f\ x\\
&(Later\ p) \gg\!= f = Later\ (p \gg\!= f)
\end{aligned}$$

We can now rewrite our semantics for expressions as a function of type $Expr \to Partial\ Int$. We ensure the definition is well-defined by making every recursive call either structurally recursive, as in the case for addition, or guarding it by *Later*, as in the case for repetition:

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Expr} \to \mathit{Partial\ Int} \\
&\mathit{eval}\ (\mathit{Val}\ n) \quad\ = \mathit{return}\ n \\
&\mathit{eval}\ (\mathit{Add}\ x\ y) = \mathbf{do}\ m \leftarrow \mathit{eval}\ x; n \leftarrow \mathit{eval}\ y; \mathit{return}\ (m + n) \\
&\mathit{eval}\ (\mathit{Repeat}\ x) = \mathbf{do}\ \mathit{eval}\ x; \mathit{Later}\ (\mathit{eval}\ (\mathit{Repeat}\ x))
\end{aligned}
$$

4.1 Compiler Specification

The compiler calculation methodology extends in a natural manner to a monadic setting [Bahr and Hutton 2022]. First of all, because the source language semantics *eval* uses the partiality monad to account for non-termination, so too does the target language semantics:

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

Our aim now is to calculate a compiler $\mathit{comp} :: \mathit{Expr} \to \mathit{Code} \to \mathit{Code}$ that satisfies the correctness property, which we formulate in a monadic style as follows:

$$\mathbf{do}\ n \leftarrow \mathit{eval}\ x; \mathit{exec}\ c\ (n : s) \quad \cong \quad \mathit{exec}\ (\mathit{comp}\ x\ c)\ s \tag{5}$$

Both sides of the equation are computations of type *Partial Stack*, and we express their equivalence using the (strong) bisimilarity relation $\cong$. Intuitively, $p \cong q$ means that either p and q both diverge, or p and q both produce the same value in the same number of steps. To formalise this idea, we define a relation $p \Downarrow_i v$ which expresses that p terminates after at most i steps with result v:

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

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

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

The semantics of the source language is defined by mixed induction and coinduction: the recursive calls for *Add* are structurally recursive, while the second recursive call for *Repeat* is guarded by *Later*. Correspondingly, in addition to structural induction over *Expr*, compiler calculation requires a coinductive reasoning principle for *Partial a* that is compatible with transitive reasoning. To this end, we define a step-indexed notion of bisimilarity. Given two computations $p, q :: \mathit{Partial}\ a$ and a natural number i, we say that p and q are i-bisimilar, written $p \cong_i q$, if the following holds:

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

Step-indexed bisimilarity is by definition downwards closed, i.e. $p \cong_i q$ implies $p \cong_j q$ for all $j \leq i$, which ensures that $\cong_i$ is a congruence for the monadic bind operator. Moreover, by definition, we have $p \cong q$ iff $p \cong_i q$ for all step counts i. Hence, our compiler correctness equation (5) can be expressed in the following equivalent form using step-indexed bisimilarity:

$$\mathbf{do}\ n \leftarrow \mathit{eval}\ x; \mathit{exec}\ c\ (n : s) \quad \cong_i \quad \mathit{exec}\ (\mathit{comp}\ x\ c)\ s \qquad \text{for all } i \tag{6}$$

We calculate a compiler from this equation by well-founded induction on lexicographically ordered pairs (i, x) of step counts i and expressions x. That is, we can assume the following induction hypothesis for all expressions x' structurally smaller than x:

$$\mathbf{do}\ n \leftarrow \mathit{eval}\ x'; \mathit{exec}\ c'\ (n : s') \quad \cong_i \quad \mathit{exec}\ (\mathit{comp}\ x'\ c')\ s'$$

In addition, we can assume that for all step counts $j < i$ and all expressions x', we have:

$$\textbf{do}\ n \leftarrow \mathit{eval}\ x'; \mathit{exec}\ c'\ (n : s') \quad \cong_j \quad \mathit{exec}\ (\mathit{comp}\ x'\ c')\ s'$$

This inductive hypothesis can then be used by applying the following proof rule, which allows the induction hypothesis to be applied to any term that is guarded by a *Later*:

$$\frac{p \cong_j q \quad \text{for all } j < i}{\mathit{Later}\ p\ \cong_i\ \mathit{Later}\ q} \tag{7}$$

During the calculation, we also use the fact that *Partial* satisfies the monad laws up to bisimilarity, and hence the laws also hold for our notion of step-indexed bisimilarity:

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

4.2 Compiler Calculation

The calculation aims to transform the left side $\textbf{do}\ n \leftarrow \mathit{eval}\ x; \mathit{exec}\ c\ (n : s)$ of equation (6) into the form $\mathit{exec}\ c'\ s$ for some code c', from which we can then define $\mathit{comp}\ x\ c = c'$ for this case. The cases for values and addition proceed in a similar manner to Section 2, except that we are now working in a monadic setting and use step-indexed bisimilarity. The *Val* calculation is given below by way of example, while the *Add* calculation can be found in Bahr and Hutton [2022]:

$$\begin{array}{ll} & \textbf{do}\ v \leftarrow \mathit{eval}\ (\mathit{Val}\ n); \mathit{exec}\ c\ (v : s) \\ = & \{\ \text{apply } \mathit{eval} \text{ for } \mathit{Val}\ \} \\ & \textbf{do}\ v \leftarrow \mathit{return}\ n; \mathit{exec}\ c\ (v : s) \\ \cong_i & \{\ \text{monad laws}\ \} \\ & \mathit{exec}\ c\ (n : s) \\ = & \{\ \text{define: } \mathit{exec}\ (\mathit{PUSH}\ n\ c)\ s = \mathit{exec}\ c\ (n : s)\ \} \\ & \mathit{exec}\ (\mathit{PUSH}\ n\ c)\ s \end{array}$$

The case for *Repeat* begins by applying definitions and the monad laws, which then allows proof rule (7) to be used to apply the induction hypothesis for all step counts $j < i$:

$$\begin{array}{ll} & \textbf{do}\ n \leftarrow \mathit{eval}\ (\mathit{Repeat}\ x); \mathit{exec}\ c\ (n : s) \\ = & \{\ \text{apply } \mathit{eval} \text{ for } \mathit{Repeat}\ \} \\ & \textbf{do}\ n \leftarrow (\textbf{do}\ \mathit{eval}\ x; \mathit{Later}\ (\mathit{eval}\ (\mathit{Repeat}\ x))); \mathit{exec}\ c\ (n : s) \\ \cong_i & \{\ \text{monad laws}\ \} \\ & \textbf{do}\ \mathit{eval}\ x; n \leftarrow \mathit{Later}\ (\mathit{eval}\ (\mathit{Repeat}\ x)); \mathit{exec}\ c\ (n : s) \\ = & \{\ \text{apply } \gg\!= \text{ for } \mathit{Later}\ \} \\ & \textbf{do}\ \mathit{eval}\ x; \mathit{Later}\ (\textbf{do}\ n \leftarrow \mathit{eval}\ (\mathit{Repeat}\ x); \mathit{exec}\ c\ (n : s)) \\ \cong_i & \{\ \text{proof rule (7) and induction hypothesis for } j < i\ \} \\ & \textbf{do}\ \mathit{eval}\ x; \mathit{Later}\ (\mathit{exec}\ (\mathit{comp}\ (\mathit{Repeat}\ x)\ c)\ s) \end{array}$$

We could now continue the calculation to obtain a term of the desired form $\mathit{exec}\ c'\ s$, from which we can then define $\mathit{comp}\ (\mathit{Repeat}\ x)\ c = c'$. However, the resulting code c' would include a recursive call to $\mathit{comp}\ (\mathit{Repeat}\ x)\ c$ itself, which means that the compiler does not terminate in this case and is hence partial. To ensure that the compiler is total, we introduce a coinductive code constructor $\mathit{REC} :: \infty\ \mathit{Code} \rightarrow \mathit{Code}$ that can be used to guard non-terminating recursive calls, similarly to the use of *Later* in the definition of *eval*. Here we use ∞ to indicate that the argument of *REC* is

coinductive, whereas arguments of the other code constructors are inductive. This is similar to the ∞ notation for mixed inductive-coinductive types in Agda [Danielsson and Altenkirch 2010], but to reduce clutter we assume conversions $\mathit{Delay} :: \mathit{Code} \to \infty\ \mathit{Code}$ and $\mathit{Force} :: \infty\ \mathit{Code} \to \mathit{Code}$ are inserted implicitly in the appropriate places as in Idris [Brady 2017].

We now resume our calculation. To ensure that the call to $\mathit{comp}\ (\mathit{Repeat}\ x)\ c$ in the current term of the calculation is guarded by *REC*, we can try to solve the following equation:

$$\mathit{exec}\ (\mathit{REC}\ (\mathit{comp}\ (\mathit{Repeat}\ x)\ c))\ s \quad = \quad \mathit{exec}\ (\mathit{comp}\ (\mathit{Repeat}\ x)\ c)\ s$$

This can be solved by generalising from the specific term $\mathit{comp}\ (\mathit{Repeat}\ x)\ c$ to give

$$\mathit{exec}\ (\mathit{REC}\ c)\ s \quad = \quad \mathit{exec}\ c\ s$$

which can then be taken as a defining clause for *exec*. However, because *REC* is a coinductive constructor, the argument c in the recursive call to *exec* is not structurally smaller than $\mathit{REC}\ c$. To ensure well-definedness, the recursive call must be guarded by the coinductive constructor *Later*. This constructor already appears in the current term in our calculation, so we can make progress by instead solving the equation

$$\mathit{exec}\ (\mathit{REC}\ (\mathit{comp}\ (\mathit{Repeat}\ x)\ c))\ s \quad = \quad \mathit{Later}\ (\mathit{exec}\ (\mathit{comp}\ (\mathit{Repeat}\ x)\ c)\ s)$$

which can be achieved by once again generalising from the specific term $\mathit{comp}\ (\mathit{Repeat}\ x)\ c$, and then taking the resulting equation as a defining clause for the function *exec*:

$$\mathit{exec}\ (\mathit{REC}\ c)\ s \quad = \quad \mathit{Later}\ (\mathit{exec}\ c\ s)$$

Using this idea, we can now continue the calculation, during which we introduce a new code constructor *POP* that simply removes the top value from the stack, motivated by the desire to transform the term into a form to which the induction hypothesis for x can be applied:

$$\begin{array}{ll}
 & \mathbf{do}\ \mathit{eval}\ x; \mathit{Later}\ (\mathit{exec}\ (\mathit{comp}\ (\mathit{Repeat}\ x)\ c)\ s) \\
= & \quad \{\ \text{define: } \mathit{exec}\ (\mathit{REC}\ c)\ s = \mathit{Later}\ (\mathit{exec}\ c\ s)\ \} \\
 & \mathbf{do}\ \mathit{eval}\ x; \mathit{exec}\ (\mathit{REC}\ (\mathit{comp}\ (\mathit{Repeat}\ x)\ c))\ s \\
= & \quad \{\ \text{define: } \mathit{exec}\ (\mathit{POP}\ c)\ (_ : s) = \mathit{exec}\ c\ s\ \} \\
 & \mathbf{do}\ n \leftarrow \mathit{eval}\ x; \mathit{exec}\ (\mathit{POP}\ (\mathit{REC}\ (\mathit{comp}\ (\mathit{Repeat}\ x)\ c)))\ (n : s) \\
\cong_i & \quad \{\ \text{induction hypothesis for } x\ \} \\
 & \mathit{exec}\ (\mathit{comp}\ x\ (\mathit{POP}\ (\mathit{REC}\ (\mathit{comp}\ (\mathit{Repeat}\ x)\ c))))\ s
\end{array}$$

The final term above now has the required form, which completes the calculation for this case. Finally, the top-level compilation function $\mathit{compile} :: \mathit{Expr} \to \mathit{Code}$ can be calculated in a similar manner to Section 2, starting from the following specification:

$$\mathbf{do}\ n \leftarrow \mathit{eval}\ x; \mathit{return}\ (n : s) \quad \cong \quad \mathit{exec}\ (\mathit{compile}\ x)\ s \tag{8}$$

Figure 3 summarises all the definitions we have calculated, along with a catch-all clause to make *exec* total, which as previously can be chosen arbitrarily as it plays no role in the calculation.

4.3 Cyclic Graphs

The compiler we have calculated for the repetition language produces linear code, but suffers from another problem that makes it unrealistic. In particular, the coinductive constructor *REC* allows code sequences to be infinite, which is utilised when compiling repetition:

$$\mathit{comp}\ (\mathit{Repeat}\ x)\ c = \mathit{comp}\ x\ (\mathit{POP}\ (\mathit{REC}\ (\mathit{comp}\ (\mathit{Repeat}\ x)\ c)))$$

However, all is not lost as the compiler only produces *cyclic sequences* of code, which can be represented finitely if we allow cyclic jumps. To this end, we define a graph-based version of the

$$
\begin{array}{l}
\textbf{data}\ \mathit{Code} = \mathit{HALT} \mid \mathit{PUSH}\ \mathit{Int}\ \mathit{Code} \mid \mathit{ADD}\ \mathit{Code} \mid \mathit{REC}\ (\infty\ \mathit{Code}) \mid \mathit{POP}\ \mathit{Code} \\
\mathit{compile} :: \mathit{Expr} \to \mathit{Code} \\
\mathit{compile}\ e = \mathit{comp}\ e\ \mathit{HALT} \\
\mathit{comp} :: \mathit{Expr} \to \mathit{Code} \to \mathit{Code} \\
\mathit{comp}\ (\mathit{Val}\ n) \quad\ c = \mathit{PUSH}\ n\ c \\
\mathit{comp}\ (\mathit{Add}\ x\ y)\ c = \mathit{comp}\ x\ (\mathit{comp}\ y\ (\mathit{ADD}\ c)) \\
\mathit{comp}\ (\mathit{Repeat}\ x)\ c = \mathit{comp}\ x\ (\mathit{POP}\ (\mathit{REC}\ (\mathit{comp}\ (\mathit{Repeat}\ x)\ c))) \\
\mathit{exec} :: \mathit{Code} \to \mathit{Stack} \to \mathit{Partial}\ \mathit{Stack} \\
\mathit{exec}\ \mathit{HALT} \quad\ s \quad\ = \mathit{return}\ s \\
\mathit{exec}\ (\mathit{PUSH}\ n\ c)\ s \quad\ = \mathit{exec}\ c\ (n : s) \\
\mathit{exec}\ (\mathit{ADD}\ c) \quad (m : n : s) = \mathit{exec}\ c\ ((n + m) : s) \\
\mathit{exec}\ (\mathit{REC}\ c) \quad s \quad\ = \mathit{Later}\ (\mathit{exec}\ c\ s) \\
\mathit{exec}\ (\mathit{POP}\ c) \quad (\_ : s) \quad = \mathit{exec}\ c\ s \\
\mathit{exec}\ \_ \quad\ \_ \quad\ = \mathit{return}\ [\,]
\end{array}
$$

Fig. 3. Initial compiler for the repetition language.

code type with explicit labels and jumps, as we did in Section 3. However, that code type only permitted acyclic control flow, i.e. code could jump forward, but not backward.

To allow backward jumps, we include a new instruction $\mathit{LAB}_g^{\leftarrow}$ that takes a function parameter of type $l \to \mathit{Code}_g\ l$, such that a term of the form $\mathit{LAB}_g^{\leftarrow}\ (\lambda l \to c)$ labels the code c with a fresh label l and makes that label available in c so that it can jump backward to itself. As we now only need to represent finite code sequences, the coinductive constructor REC is no longer required. Taken together, our graph-based code type is defined as follows:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Code}_g\ l = \mathit{HALT}_g \mid \mathit{PUSH}_g\ \mathit{Int}\ (\mathit{Code}_g\ l) \mid \mathit{ADD}_g\ (\mathit{Code}_g\ l) \mid \mathit{POP}_g\ (\mathit{Code}_g\ l) \mid \mathit{JMP}_g\ l \mid \\
\qquad\qquad \mathit{LAB}_g^{\rightarrow}\ (l \to \mathit{Code}_g\ l)\ (\mathit{Code}_g\ l) \mid \mathit{LAB}_g^{\leftarrow}\ (l \to \mathit{Code}_g\ l)
\end{array}
$$

For example, the code fragment from the beginning of this section,

```
      PUSH 0
l :   PUSH 2
      ADD
      JMP l
```

can be represented by the following term of type $\mathit{Code}_g\ l$:

$$\mathit{PUSH}_g\ 0\ (\mathit{LAB}_g^{\leftarrow}\ (\lambda l \to \mathit{PUSH}_g\ 2\ (\mathit{ADD}_g\ (\mathit{JMP}_g\ l))))$$

The function $(\!|\cdot|\!)$ that unravels a code graph to a code sequence is then defined as follows:

$$
\begin{array}{l}
(\!|\cdot|\!) :: \mathit{Code}_g\ \mathit{Code} \to \mathit{Code} \\
(\!|\mathit{HALT}_g|\!) \quad\ = \mathit{HALT} \\
(\!|\mathit{PUSH}_g\ n\ c|\!) = \mathit{PUSH}\ n\ (\!|c|\!) \\
(\!|\mathit{ADD}_g\ c|\!) \quad\ = \mathit{ADD}\ (\!|c|\!) \\
(\!|\mathit{POP}_g\ c|\!) \quad\ = \mathit{POP}\ (\!|c|\!) \\
(\!|\mathit{JMP}_g\ l|\!) \quad\ = l \\
(\!|\mathit{LAB}_g^{\rightarrow}\ f\ c|\!) \ = (\!|f\ (\!|c|\!)|\!) \\
(\!|\mathit{LAB}_g^{\leftarrow}\ f|\!) \quad\ = (\!|f\ (\mathit{REC}\ (\!|\mathit{LAB}_g^{\leftarrow}\ f|\!))|\!)
\end{array}
$$

Note that for a backward jump $LAB_g^{\leftarrow}$ one of the recursive calls is not structurally recursive, and hence we need to guard it by the coinductive constructor *REC*. For instance, applying $(\!|\cdot|\!)$ to the example code graph above gives the following infinite, but cyclic, code sequence:

$PUSH\ 0\ c$ **where** $c = PUSH\ 2\ (ADD\ (REC\ c))$

4.4 Compiler Calculation

We can now calculate a graph-based compiler for the repetition language in a similar manner to Section 3. As previously, we begin by defining the semantics of code graphs by unravelling:

$exec_g :: (\forall\, l.\ Code_g\ l) \to Stack \to Partial\ Stack$
$exec_g\ c\ s = exec\ (\!|c|\!)\ s$

The correctness of a graph-based compilation function $comp_g :: Expr \to Code_g\ l \to Code_g\ l$ can then be expressed by the following bisimilarity equation:

$$\mathbf{do}\ n \leftarrow eval\ x; exec_g\ c\ (n : s) \quad \cong \quad exec_g\ (comp_g\ x\ c)\ s \tag{9}$$

Similarly to the approach in Section 3, we can use the definition of $exec_g$ and the correctness equation (5) for the original compiler *comp* to strengthen this equation to:

$$comp\ x\ (\!|c|\!) \quad \cong \quad (\!|comp_g\ x\ c|\!) \tag{10}$$

This equation uses a bisimilarity relation $\cong$ on the *Code* type. Similarly to bisimilarity on the coinductive type *Partial*, bisimilarity on code can be expressed equivalently by a step-indexed relation $\cong_i$, where for two pieces of code $c, d :: Code$, writing $c \cong_i d$ intuitively means that we cannot distinguish between c and d if we traverse both code sequences but may only look under at most i nested occurrences of the coinductive constructor *REC*. Hence, equation (10) can be expressed in the following equivalent form using step-indexed bisimilarity:

$$comp\ x\ (\!|c|\!) \quad \cong_i \quad (\!|comp_g\ x\ c|\!) \qquad \text{for all } i \tag{11}$$

The step-indexed relation $\cong_i$ is a congruence for all inductive constructors of the *Code* type. In addition, in a similar manner to the coinductive constructor *Later* of the *Partial* monad, we have the following proof rule for the coinductive constructor *REC*:

$$\frac{c \cong_j d \qquad \text{for all } j < i}{REC\ c \quad \cong_i \quad REC\ d} \tag{12}$$

From the above congruence properties, we also obtain the following congruence property for *comp*, which can be proved by a straightforward induction on the expression x:

$$\frac{c \quad \cong_i \quad d}{comp\ x\ c \quad \cong_i \quad comp\ x\ d} \tag{13}$$

Calculation of the compiler $comp_g$ now proceeds from equation (11) by well-founded induction on lexicographically ordered pairs (i, x). For each case, we aim to transform the left side of the equation, $comp\ x\ (\!|c|\!)$, into the form $(\!|c'|\!)$ for some code c', so that we can then define $comp_g\ x\ c = c'$. The calculations for values and addition are as in Section 3. The calculation for repetition is given below. Its key step is abstracting over the term starting with *REC* that is motivated by the desire to apply the definition clause of $(\!|\cdot|\!)$ in which such a term appears as argument to a function.

$$\begin{aligned}
& comp\ (Repeat\ x)\ (\!|c|\!) \\
= \quad & \{\ \text{apply } comp \text{ for } Repeat\ \} \\
& comp\ x\ (POP\ (REC\ (comp\ (Repeat\ x)\ (\!|c|\!))))
\end{aligned}$$

$$
\begin{array}{ll}
\cong_i & \{ \text{ proof rule (11) and induction hypothesis for } j < i \text{ } \} \\
 & \mathit{comp}\ x\ (\mathit{POP}\ (\mathit{REC}\ (\!|\mathit{comp}_g\ (\mathit{Repeat}\ x)\ c|\!))) \\
= & \{ \text{ abstract over } \mathit{REC}\ (\!|\mathit{comp}_g\ (\mathit{Repeat}\ x)\ c|\!) \text{ } \} \\
 & (\lambda l \to \mathit{comp}\ x\ (\mathit{POP}\ l))\ (\mathit{REC}\ (\!|\mathit{comp}_g\ (\mathit{Repeat}\ x)\ c|\!)) \\
= & \{ \text{ unapply } (\!|\cdot|\!) \text{ for } \mathcal{J}\mathit{MP}_g \text{ } \} \\
 & (\lambda l \to \mathit{comp}\ x\ (\mathit{POP}\ (\!|\mathcal{J}\mathit{MP}_g\ l|\!)))\ (\mathit{REC}\ (\!|\mathit{comp}_g\ (\mathit{Repeat}\ x)\ c|\!)) \\
= & \{ \text{ unapply } (\!|\cdot|\!) \text{ for } \mathit{POP}_g \text{ } \} \\
 & (\lambda l \to \mathit{comp}\ x\ (\!|\mathit{POP}_g\ (\mathcal{J}\mathit{MP}_g\ l)|\!))\ (\mathit{REC}\ (\!|\mathit{comp}_g\ (\mathit{Repeat}\ x)\ c|\!)) \\
\cong_i & \{ \text{ induction hypothesis for } x \text{ } \} \\
 & (\lambda l \to (\!|\mathit{comp}_g\ x\ (\mathit{POP}_g\ (\mathcal{J}\mathit{MP}_g\ l))|\!))\ (\mathit{REC}\ (\!|\mathit{comp}_g\ (\mathit{Repeat}\ x)\ c|\!)) \\
= & \{ \text{ unapply } (\!|\cdot|\!) \text{ for } \mathit{LAB}_g^{\leftarrow} \text{ } \} \\
 & (\!|\mathit{LAB}_g^{\leftarrow}\ (\lambda l \to \mathit{comp}_g\ x\ (\mathit{POP}_g\ (\mathcal{J}\mathit{MP}_g\ l)))|\!)
\end{array}
$$

The final term is now of the required form, which gives the following clause for comp_g:

$$\mathit{comp}_g\ (\mathit{Repeat}\ x)\ c = \mathit{LAB}_g^{\leftarrow}\ (\lambda l \to \mathit{comp}_g\ x\ (\mathit{POP}_g\ (\mathcal{J}\mathit{MP}_g\ l)))$$

To understand the final step of the calculation, it is helpful to expand out the term $\mathit{comp}_g\ (\mathit{Repeat}\ x)\ c$ using this clause, from which it is then clear that the final step uses the definition of $(\!|\cdot|\!)$ for backward labels, in the beta-expanded form $(\!|\mathit{LAB}_g^{\leftarrow}\ f|\!) = (\lambda l \to (\!|f\ l|\!))\ (\mathit{REC}\ (\!|\mathit{LAB}_g^{\leftarrow}\ f|\!))$:

$$
\begin{array}{ll}
 & (\lambda l \to (\!|\mathit{comp}_g\ x\ (\mathit{POP}_g\ (\mathcal{J}\mathit{MP}_g\ l))|\!))\ (\mathit{REC}\ (\!|\mathit{LAB}_g^{\leftarrow}\ (\lambda l \to \mathit{comp}_g\ x\ (\mathit{POP}_g\ (\mathcal{J}\mathit{MP}_g\ l)))|\!)) \\
= & \{ \text{ unapply } (\!|\cdot|\!) \text{ for } \mathit{LAB}_g^{\leftarrow} \text{ } \} \\
 & (\!|\mathit{LAB}_g^{\leftarrow}\ (\lambda l \to \mathit{comp}_g\ x\ (\mathit{POP}_g\ (\mathcal{J}\mathit{MP}_g\ l)))|\!)
\end{array}
$$

Figure 4 gives the full definition for comp_g that we have calculated, along with the top-level compilation function $\mathit{compile}_g$ that can be calculated in a similar manner to previously.

In conclusion, we have derived a graph-based compiler that produces finite code by representing the infinite code sequences produced by *compile* using backward jumps. For example, applying $\mathit{compile}_g$ to the expression $\mathit{Repeat}\ (\mathit{Add}\ (\mathit{Val}\ 2)\ (\mathit{Val}\ 3))$ gives the code graph

$$\mathit{LAB}_g^{\leftarrow}\ (\lambda l \to \mathit{PUSH}_g\ 2\ (\mathit{PUSH}_g\ 3\ (\mathit{ADD}_g\ (\mathit{POP}_g\ (\mathcal{J}\mathit{MP}_g\ l)))))$$

which corresponds to the following code in assembly-like notation:

```
l:  PUSH 2
    PUSH 3
    ADD
    POP
    JMP l
```

In particular, this code avoids the need for infinite code by using a label and jump to capture the cyclic behaviour of the repetition primitive in the source language.

5 While Loops and State

For our final example, we consider a more realistic language that features while loops, along with primitives for reading and writing a mutable reference cell that stores an integer value. In particular, the reference cell allows loops to be constructed that have any desired number of iterations, both finite and infinite. The syntax of the language is defined as follows:

data *Expr* = *Val Int* | *Add Expr Expr* | *Get* | *Put Expr Expr* | *While Expr Expr*

Informally, *Get* returns the current value of the reference cell, *Put x y* sets the cell to the value of the expression x and then behaves as the expression y, and *While x y* repeatedly evaluates y as

$$
\begin{array}{l}
\mathit{compile}_g :: \mathit{Expr} \to \mathit{Code}_g\ l \\
\mathit{compile}_g\ x = \mathit{comp}_g\ x\ \mathit{HALT}_g \\
\mathit{comp}_g :: \mathit{Expr} \to \mathit{Code}_g\ l \to \mathit{Code}_g\ l \\
\mathit{comp}_g\ (\mathit{Val}\ n) \quad\ \ c = \mathit{PUSH}_g\ n\ c \\
\mathit{comp}_g\ (\mathit{Add}\ x\ y)\ \ c = \mathit{comp}_g\ x\ (\mathit{comp}_g\ y\ (\mathit{ADD}_g\ c)) \\
\mathit{comp}_g\ (\mathit{Repeat}\ x)\ c = \mathit{LAB}_g^{\leftarrow}\ (\lambda l \to \mathit{comp}_g\ x\ (\mathit{POP}_g\ (\mathit{JMP}_g\ l)))
\end{array}
$$

Fig. 4. Graph-based compiler for the repetition language.

long as x has a non-zero value. For example, the following expression sets the reference cell to ten and then repeatedly decrements the reference until it becomes zero:

$$\mathit{Put}\ (\mathit{Val}\ 10)\ (\mathit{While}\ \mathit{Get}\ (\mathit{Put}\ (\mathit{Add}\ \mathit{Get}\ (\mathit{Val}\ (-1)))\ \mathit{Get}))$$

Because while loops may not always terminate, we again use the partiality monad to define the semantics of the language. In addition, we also need to keep track of the state of the reference cell, so the semantics of an expression becomes a state transformer:

$$
\begin{array}{l}
\textbf{type}\ \mathit{State} = \mathit{Int} \\
\mathit{eval} :: \mathit{Expr} \to \mathit{State} \to \mathit{Partial}\ (\mathit{Int}, \mathit{State}) \\
\mathit{eval}\ (\mathit{Val}\ n) \qquad q = \mathit{return}\ (n, q) \\
\mathit{eval}\ (\mathit{Add}\ x\ y) \quad q = \textbf{do}\ (m, q_1) \leftarrow \mathit{eval}\ x\ q; (n, q_2) \leftarrow \mathit{eval}\ y\ q_1; \mathit{return}\ (m + n, q_2) \\
\mathit{eval}\ \mathit{Get} \qquad\qquad q = \mathit{return}\ (q, q) \\
\mathit{eval}\ (\mathit{Put}\ x\ y) \quad q = \textbf{do}\ (n, \_) \leftarrow \mathit{eval}\ x\ q; \mathit{eval}\ y\ n \\
\mathit{eval}\ (\mathit{While}\ x\ y)\ q = \textbf{do}\ (n, q_1) \leftarrow \mathit{eval}\ x\ q \\
\qquad\qquad\qquad\qquad \textbf{if}\ n == 0\ \textbf{then}\ \mathit{return}\ (0, q_1) \\
\qquad\qquad\qquad\qquad \textbf{else do}\ (\_, q_2) \leftarrow \mathit{eval}\ y\ q_1; \mathit{Later}\ (\mathit{eval}\ (\mathit{While}\ x\ y)\ q_2)
\end{array}
$$

As we already use the symbol s for stacks, in this section we use the symbol q for states. Note that the above semantics combines the partiality monad with the state monad. However, for calculation purposes we don't treat the state monad abstractly, but instead explicitly manipulate the state as shown above. This approach allows us to discover an appropriate mechanism to compile the state monad effect to a lower-level implementation in the target machine [Bahr and Hutton 2015].

5.1 Tree-Based Compiler

For the tree-based compiler, we follow the methodology of Bahr and Hutton [2015], in which the target machine is specified as a function $\mathit{exec} :: \mathit{Code} \to \mathit{Conf} \to \mathit{Conf}$, where Conf is the type of machine configurations that comprise a stack and a state. However, as in the previous section, to account for non-termination we amend the type of exec to use the partiality monad:

$$
\begin{array}{l}
\textbf{type}\ \mathit{Conf} = (\mathit{Stack}, \mathit{State}) \\
\mathit{exec} :: \mathit{Code} \to \mathit{Conf} \to \mathit{Partial}\ \mathit{Conf}
\end{array}
$$

The type of the compiler remains the same as previously, $\mathit{comp} :: \mathit{Expr} \to \mathit{Code} \to \mathit{Code}$, while the compiler correctness property can be adapted for the presence of state as follows:

$$\textbf{do}\ (n, q') \leftarrow \mathit{eval}\ x\ q; \mathit{exec}\ c\ (n : s, q') \quad \cong \quad \mathit{exec}\ (\mathit{comp}\ x\ c)\ (s, q) \tag{14}$$

As before, however, for the purposes of calculating the compiler we utilise the following equivalent version that is expressed using step-indexed bisimilarity:

$$\mathbf{do}\ (n, q') \leftarrow \mathit{eval}\ x\ q; \mathit{exec}\ c\ (n : s, q') \quad \cong_i \quad \mathit{exec}\ (\mathit{comp}\ x\ c)\ (s, q) \qquad \text{for all } i \tag{15}$$

We now proceed with the calculation, by well-founded induction on lexicographically ordered pairs (i, x). In a similar manner to previously, we start with the left side of equation (15) and aim to transform it into the form $\mathit{exec}\ c'\ (s, q)$ for some code c', so that we can then define $\mathit{comp}\ x\ c = c'$. We give the calculations for *Get* and *While* below by way of example. The other cases are similar and can be found in the accompanying Agda formalisation.

The case for *Get* applies the definition of *eval*, simplifies the resulting term using the monad laws, and finally introduces a new code constructor *LOAD* to obtain a term of the desired form:

$$\begin{array}{ll}
 & \mathbf{do}\ (n, q') \leftarrow \mathit{eval}\ \mathit{Get}\ q; \mathit{exec}\ c\ (n : s, q') \\
= & \quad \{\text{ apply } \mathit{eval} \text{ for } \mathit{Get} \} \\
 & \mathbf{do}\ (n, q') \leftarrow \mathit{return}\ (q, q); \mathit{exec}\ c\ (n : s, q') \\
\cong_i & \quad \{\text{ monad laws } \} \\
 & \mathit{exec}\ c\ (q : s, q) \\
= & \quad \{\text{ define } \mathit{exec}\ (\mathit{LOAD}\ c)\ (s, q) = \mathit{exec}\ c\ (q : s, q) \} \\
 & \mathit{exec}\ (\mathit{LOAD}\ c)\ (s, q)
\end{array}$$

The case for *While* begins in a manner similar to *Repeat* from Section 4.2:

$$\begin{array}{ll}
 & \mathbf{do}\ (m, q') \leftarrow \mathit{eval}\ (\mathit{While}\ x\ y)\ q; \mathit{exec}\ c\ (m : s, q') \\
= & \quad \{\text{ apply } \mathit{eval} \text{ for } \mathit{While} \} \\
 & \mathbf{do}\ (m, q') \leftarrow (\mathbf{do}\ (n, q_1) \leftarrow \mathit{eval}\ x\ q; \mathbf{if}\ n == 0\ \mathbf{then}\ \mathit{return}\ (0, q_1) \\
 & \qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else\ do}\ (_, q_2) \leftarrow \mathit{eval}\ y\ q_1 \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Later}\ (\mathit{eval}\ (\mathit{While}\ x\ y)\ q_2)) \\
 & \quad \mathit{exec}\ c\ (m : s, q') \\
\cong_i & \quad \{\text{ monad laws } \} \\
 & \mathbf{do}\ (n, q_1) \leftarrow \mathit{eval}\ x\ q; \mathbf{if}\ n == 0\ \mathbf{then}\ \mathit{exec}\ c\ (0 : s, q_1) \\
 & \qquad\qquad\qquad \mathbf{else\ do}\ (_, q_2) \leftarrow \mathit{eval}\ y\ q_1 \\
 & \qquad\qquad\qquad\qquad (n, q_3) \leftarrow \mathit{Later}\ (\mathit{eval}\ (\mathit{While}\ x\ y)\ q_2) \\
 & \qquad\qquad\qquad\qquad \mathit{exec}\ c\ (n : s, q_3) \\
= & \quad \{\text{ apply } \gg\!= \text{ for } \mathit{Later} \} \\
 & \mathbf{do}\ (n, q_1) \leftarrow \mathit{eval}\ x\ q; \mathbf{if}\ n == 0\ \mathbf{then}\ \mathit{exec}\ c\ (0 : s, q_1) \\
 & \qquad\qquad\qquad \mathbf{else\ do}\ (_, q_2) \leftarrow \mathit{eval}\ y\ q_1 \\
 & \qquad\qquad\qquad\qquad \mathit{Later}\ (\mathbf{do}\ (n, q_3) \leftarrow \mathit{eval}\ (\mathit{While}\ x\ y)\ q_2; \mathit{exec}\ c\ (n : s, q_3)) \\
\cong_i & \quad \{\text{ proof rule (7) and induction hypothesis for } j < i \} \\
 & \mathbf{do}\ (n, q_1) \leftarrow \mathit{eval}\ x\ q; \mathbf{if}\ n == 0\ \mathbf{then}\ \mathit{exec}\ c\ (0 : s, q_1) \\
 & \qquad\qquad\qquad \mathbf{else\ do}\ (_, q_2) \leftarrow \mathit{eval}\ y\ q_1 \\
 & \qquad\qquad\qquad\qquad \mathit{Later}\ (\mathit{exec}\ (\mathit{comp}\ (\mathit{While}\ x\ y)\ c)\ (s, q_2))
\end{array}$$

Similarly to the calculation for *Repeat*, to ensure that the resulting compiler is total, we now seek to guard the call $\mathit{comp}\ (\mathit{While}\ x\ y)\ c$ in the final term above by a coinductive code constructor $\mathit{REC} :: \infty\ \mathit{Code} \rightarrow \mathit{Code}$. That is, we seek to solve the following equation:

$$\mathit{exec}\ (\mathit{REC}\ (\mathit{comp}\ (\mathit{While}\ x\ y)\ c))\ (s, q_2) \quad = \quad \mathit{Later}\ (\mathit{exec}\ (\mathit{comp}\ (\mathit{While}\ x\ y)\ c)\ (s, q_2))$$

This can be solved by generalising from the specific term $comp\ (While\ x\ y)\ c$ to give

$$exec\ (REC\ c)\ (s, q_2) \quad = \quad Later\ (exec\ c\ (s, q_2))$$

which can then be taken as defining clause for *exec*. Using this idea, we can now continue the calculation, during which we introduce new code constructors *POP* and *JPZ* in a similar manner to previously, to allow the induction hypotheses for the argument expressions to be applied:

$$\begin{array}{ll}
 & \mathbf{do}\ (n, q_1) \leftarrow eval\ x\ q; \mathbf{if}\ n == 0\ \mathbf{then}\ exec\ c\ (0 : s, q_1) \\
 & \qquad\qquad\qquad \mathbf{else\ do}\ (_, q_2) \leftarrow eval\ y\ q_1; Later\ (exec\ (comp\ (While\ x\ y)\ c)\ (s, q_2)) \\
= & \{\ \text{define: } exec\ (REC\ c)\ (s, q) = Later\ (exec\ c\ (s, q))\ \} \\
 & \mathbf{do}\ (n, q_1) \leftarrow eval\ x\ q; \mathbf{if}\ n == 0\ \mathbf{then}\ exec\ c\ (0 : s, q_1) \\
 & \qquad\qquad\qquad \mathbf{else\ do}\ (_, q_2) \leftarrow eval\ y\ q_1; exec\ (REC\ (comp\ (While\ x\ y)\ c))\ (s, q_2) \\
= & \{\ \text{define: } exec\ (POP\ c)\ (_ : s, q) = exec\ c\ (s, q)\ \} \\
 & \mathbf{do}\ (n, q_1) \leftarrow eval\ x\ q; \mathbf{if}\ n == 0\ \mathbf{then}\ exec\ c\ (0 : s, q_1) \\
 & \qquad\qquad\qquad \mathbf{else\ do}\ (m, q_2) \leftarrow eval\ y\ q_1 \\
 & \qquad\qquad\qquad\qquad exec\ (POP\ (REC\ (comp\ (While\ x\ y)\ c)))\ (m : s, q_2) \\
\cong_i & \{\ \text{induction hypothesis for } y\ \} \\
 & \mathbf{do}\ (n, q_1) \leftarrow eval\ x\ q; \mathbf{if}\ n == 0\ \mathbf{then}\ exec\ c\ (0 : s, q_1) \\
 & \qquad\qquad\qquad \mathbf{else}\ exec\ (comp\ y\ (POP\ (REC\ (comp\ (While\ x\ y)\ c))))\ (s, q_1) \\
= & \{\ \text{define: } exec\ (JPZ\ c'\ c)\ (n : s, q) = \mathbf{if}\ n == 0\ \mathbf{then}\ exec\ c'\ (0 : s, q)\ \mathbf{else}\ exec\ c\ (s, q)\ \} \\
 & \mathbf{do}\ (n, q_1) \leftarrow eval\ x\ q; exec\ JPZ\ c\ (comp\ y\ (POP\ (REC\ (comp\ (While\ x\ y)\ c))))\ (n : s, q_1) \\
\cong_i & \{\ \text{induction hypothesis for } x\ \} \\
 & exec\ (comp\ x\ (JPZ\ c\ (comp\ y\ (POP\ (REC\ (comp\ (While\ x\ y)\ c))))))\ (s, q)
\end{array}$$

The final term now has the required form, from which we obtain the definition:

$$comp\ (While\ x\ y)\ c = comp\ x\ (JPZ\ c\ (comp\ y\ (POP\ (REC\ (comp\ (While\ x\ y)\ c)))))$$

Figure 5 presents all definitions that are calculated in this manner, along with the top-level compilation function *compile*, and a catch-all clause for *exec* to make it total.

5.2 Graph-Based Compiler

The compiler we have calculated for the repetition language is unrealistic for three reasons, all concerning while loops. First of all, the compiler produces tree-shaped code because the *JPZ* instruction takes two pieces of code as arguments. Secondly, it duplicates the additional code c when compiling while loops. And finally, it produces infinite code because while loops are compiled by unrolling the loop. However, the compiler in fact only produces *regular trees*, i.e. trees with only finitely many subtrees, which may be represented finitely if we allow cyclic jumps.

To calculate a graph-based compiler from the tree-based version in Figure 5, we proceed in the same manner as the repetition language in the previous section. First of all, we turn the *Code* type into a corresponding structured graph type $Code_g\ l$ that supports forward and backward jumps, and define its canonical unravelling function $(\!|\cdot|\!)$, as shown in Figure 6.

Next, we define the semantics of the graph-based target language by unravelling:

$$\begin{array}{l}
exec_g :: (\forall\, l.\ Code_g\ l) \to Conf \to Partial\ Conf \\
exec_g\ c\ s = exec\ (\!|c|\!)\ s
\end{array}$$

The correctness of our graph-based compilation function $comp_g :: Expr \to Code_g\ l \to Code_g\ l$ can then be expressed by the following bisimilarity relation in the partiality monad:

$$exec_g\ c\ (eval\ x : s, q) \quad \cong \quad exec_g\ (comp_g\ x\ c)\ (s, q) \tag{16}$$

```
data Code = HALT | PUSH Int Code | ADD Code | LOAD Code | STORE Code |
            REC (∞ Code) | POP Code | JPZ Code Code

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

comp :: Expr → Code → Code
comp (Val n)       c = PUSH n c
comp (Add x y)     c = comp x (comp y (ADD c))
comp Get           c = LOAD c
comp (Put x y)     c = comp x (STORE (comp y c))
comp (While x y) c = comp x (JPZ c (comp y (POP (REC (comp (While x y) c)))))

exec :: Code → Conf → Partial Conf
exec HALT       conf        = return conf
exec (PUSH n c) (s, q)      = exec c (n : s, q)
exec (ADD c)    (m : n : s, q) = exec c ((n + m) : s, q)
exec (LOAD c)   (s, q)      = exec c (q : s, q)
exec (STORE c)  (n : s, _)  = exec c (s, n)
exec (REC c)    conf        = Later (exec c conf)
exec (POP c)    (_ : s, q)  = exec c (s, q)
exec (JPZ c′ c) (n : s, q)  = if n == 0 then exec c′ (0 : s, q) else exec c (s, q)
exec _          _           = return ([ ], 0)
```

Fig. 5. Tree-based compiler for the while language.

```
data Code_g l                                  (|·|) :: Code_g Code → Code
  = HALT_g                                     (|HALT_g|)       = HALT
  | PUSH_g Int (Code_g l)                      (|PUSH_g n c|)   = PUSH n (|c|)
  | ADD_g (Code_g l)                           (|ADD_g c|)      = ADD (|c|)
  | LOAD_g (Code_g l)                          (|LOAD_g c|)     = LOAD (|c|)
  | STORE_g (Code_g l)                         (|STORE_g c|)    = STORE (|c|)
  | POP_g (Code_g l)                           (|POP_g c|)      = POP (|c|)
  | JPZ_g l (Code_g l)                         (|JPZ_g l c|)    = JPZ l (|c|)
  | JMP_g l                                    (|JMP_g l|)      = l
  | LAB_g^→ (l → Code_g l) (Code_g l)          (|LAB_g^→ f c|)  = (|f (|c|)|)
  | LAB_g^← (l → Code_g l)                     (|LAB_g^← f|)    = (|f (REC (|LAB_g^← f|))|)
```

Fig. 6. Graph-based code and its unravelling for the while language.

Using the definition of $exec_g$ and the correctness equation (14) for the tree-based compiler *comp*, this equation can be strengthened to a bisimilarity relation on the *Code* type,

$$comp\ x\ (\!|c|\!) \quad \cong \quad (\!|comp_g\ x\ c|\!) \tag{17}$$

which can then be formulated equivalently using a step-indexed bisimilarity relation on *Code*:

$$comp\ x\ (\!|c|\!) \quad \cong_i \quad (\!|comp_g\ x\ c|\!) \qquad \text{for all } i \tag{18}$$

$$
\begin{array}{ll}
 & \mathit{comp}\ (\mathit{While}\ x\ y)\ (\!|c|\!) \\
= & \quad \{\ \text{apply } \mathit{comp} \text{ for } \mathit{While}\ \} \\
 & \mathit{comp}\ x\ (\mathcal{J}\mathit{PZ}\ (\!|c|\!)\ (\mathit{comp}\ y\ (\mathit{POP}\ (\mathit{REC}\ (\mathit{comp}\ (\mathit{While}\ x\ y)\ (\!|c|\!))))) \\
\cong_i & \quad \{\ \text{induction hypothesis for } j < i\ \} \\
 & \mathit{comp}\ x\ (\mathcal{J}\mathit{PZ}\ (\!|c|\!)\ (\mathit{comp}\ y\ (\mathit{POP}\ (\mathit{REC}\ (\!|\mathit{comp}_g\ (\mathit{While}\ x\ y)\ c|\!)))) \\
= & \quad \{\ \text{abstract over } \mathit{REC}\ (\!|\mathit{comp}_g\ (\mathit{While}\ x\ y)\ c|\!)\ \} \\
 & (\lambda l \to \mathit{comp}\ x\ (\mathcal{J}\mathit{PZ}\ (\!|c|\!)\ (\mathit{comp}\ y\ (\mathit{POP}\ l))))\ (\mathit{REC}\ (\!|\mathit{comp}_g\ (\mathit{While}\ x\ y)\ c|\!)) \\
= & \quad \{\ \text{unapply } (\!|\cdot|\!) \text{ for } \mathcal{J}\mathit{MP}_g\ \} \\
 & (\lambda l \to \mathit{comp}\ x\ (\mathcal{J}\mathit{PZ}\ (\!|c|\!)\ (\mathit{comp}\ y\ (\mathit{POP}\ (\!|\mathcal{J}\mathit{MP}_g\ l|\!)))))\ (\mathit{REC}\ (\!|\mathit{comp}_g\ (\mathit{While}\ x\ y)\ c|\!)) \\
= & \quad \{\ \text{unapply } (\!|\cdot|\!) \text{ for } \mathit{POP}_g\ \} \\
 & (\lambda l \to \mathit{comp}\ x\ (\mathcal{J}\mathit{PZ}\ (\!|c|\!)\ (\mathit{comp}\ y\ (\!|\mathit{POP}_g\ (\mathcal{J}\mathit{MP}_g\ l)|\!))))\ (\mathit{REC}\ (\!|\mathit{comp}_g\ (\mathit{While}\ x\ y)\ c|\!)) \\
\cong_i & \quad \{\ \text{induction hypothesis for } y\ \} \\
 & (\lambda l \to \mathit{comp}\ x\ (\mathcal{J}\mathit{PZ}\ (\!|c|\!)\ (\!|\mathit{comp}_g\ y\ (\mathit{POP}_g\ (\mathcal{J}\mathit{MP}_g\ l))|\!)))\ (\mathit{REC}\ (\!|\mathit{comp}_g\ (\mathit{While}\ x\ y)\ c|\!)) \\
= & \quad \{\ \text{abstract over } (\!|c|\!)\ \} \\
 & (\lambda l \to \mathit{comp}\ x\ ((\lambda l' \to \mathcal{J}\mathit{PZ}\ l'\ (\!|\mathit{comp}_g\ y\ (\mathit{POP}_g\ (\mathcal{J}\mathit{MP}_g\ l))|\!))\ (\!|c|\!)))\ (\mathit{REC}\ (\!|\mathit{comp}_g\ (\mathit{While}\ x\ y)\ c|\!)) \\
= & \quad \{\ \text{unapply } (\!|\cdot|\!) \text{ for } \mathcal{J}\mathit{PZ}_g\ \} \\
 & (\lambda l \to \mathit{comp}\ x\ ((\lambda l' \to (\!|\mathcal{J}\mathit{PZ}_g\ l'\ (\mathit{comp}_g\ y\ (\mathit{POP}_g\ (\mathcal{J}\mathit{MP}_g\ l)))|\!))\ (\!|c|\!)))\ (\mathit{REC}\ (\!|\mathit{comp}_g\ (\mathit{While}\ x\ y)\ c|\!)) \\
= & \quad \{\ \text{unapply } (\!|\cdot|\!) \text{ for } \mathit{LAB}_g^{\rightarrow}\ \} \\
 & (\lambda l \to \mathit{comp}\ x\ (\!|\mathit{LAB}_g^{\rightarrow}\ (\lambda l' \to \mathcal{J}\mathit{PZ}_g\ l'\ (\mathit{comp}_g\ y\ (\mathit{POP}_g\ (\mathcal{J}\mathit{MP}_g\ l))))\ c|\!))\ (\mathit{REC}\ (\!|\mathit{comp}_g\ (\mathit{While}\ x\ y)\ c|\!)) \\
\cong_i & \quad \{\ \text{induction hypothesis for } x\ \} \\
 & (\lambda l \to (\!|\mathit{comp}_g\ x\ (\mathit{LAB}_g^{\rightarrow}\ (\lambda l' \to \mathcal{J}\mathit{PZ}_g\ l'\ (\mathit{comp}_g\ y\ (\mathit{POP}_g\ (\mathcal{J}\mathit{MP}_g\ l))))\ c)|\!))\ (\mathit{REC}\ (\!|\mathit{comp}_g\ (\mathit{While}\ x\ y)\ c|\!)) \\
= & \quad \{\ \text{unapply } (\!|\cdot|\!) \text{ for } \mathit{LAB}_g^{\leftarrow}\ \} \\
 & (\!|\mathit{LAB}_g^{\leftarrow}\ (\lambda l \to \mathit{comp}_g\ x\ (\mathit{LAB}_g^{\rightarrow}\ (\lambda l' \to \mathcal{J}\mathit{PZ}_g\ l'\ (\mathit{comp}_g\ y\ (\mathit{POP}_g\ (\mathcal{J}\mathit{MP}_g\ l))))\ c))|\!)
\end{array}
$$

Fig. 7. Calculation of comp_g for while loops.

The calculation of comp_g proceeds from equation (18) by induction on lexicographically ordered pairs (i, x). The cases for *Val*, *Add*, *Get* and *Put* are straightforward: first apply the definition of *comp*, then turn all tree-based terms into graph-based versions in a syntax-directed manner by applying the definition of unravelling, or an induction hypothesis. The interesting case is the calculation for *While*, which is shown in Figure 7. This calculation combines the ideas that were used for conditionals in Section 3 and repetition in Section 4. In particular, the key steps in the calculation are abstracting over the *REC* term to introduce a backward jump that avoids infinite code, and abstracting over the term $(\!|c|\!)$ to introduce a forward jump that avoids tree-shaped code and code duplication. The result of the calculation is the graph-based compiler implementation shown in Figure 8.

6 Related Work

We have discussed related work on deriving correct-by-construction compilers throughout the paper. We supplement that discussion with related work in other relevant areas below.

Graphs. There are many approaches to representing graph in functional languages. Most use explicit names to identify vertices in a graph, often represented as integers [Erwig 2001; Kashiwagi and Wise 1991; King and Launchbury 1995]. Such representations are often motivated by efficiency considerations. However, efficient graph representations can still admit powerful high-level reasoning principles as King and Launchbury [1995] have shown. More recently, Mokhov

$$
\begin{array}{l}
\mathit{compile}_g :: \mathit{Expr} \to \mathit{Code}_g\ l \\
\mathit{compile}_g\ e = \mathit{comp}_g\ e\ \mathit{HALT}_g \\
\\
\mathit{comp}_g :: \mathit{Expr} \to \mathit{Code}_g\ l \to \mathit{Code}_g\ l \\
\mathit{comp}_g\ (\mathit{Val}\ n) \qquad c = \mathit{PUSH}_g\ n\ c \\
\mathit{comp}_g\ (\mathit{Add}\ x\ y) \quad c = \mathit{comp}_g\ x\ (\mathit{comp}_g\ y\ (\mathit{ADD}_g\ c)) \\
\mathit{comp}_g\ \mathit{Get} \qquad\qquad c = \mathit{LOAD}_g\ c \\
\mathit{comp}_g\ (\mathit{Put}\ x\ y) \quad c = \mathit{comp}_g\ x\ (\mathit{STORE}_g\ (\mathit{comp}_g\ y\ c)) \\
\mathit{comp}_g\ (\mathit{While}\ x\ y)\ c = \mathit{LAB}_g^{\leftarrow}\ (\lambda l \to \mathit{comp}_g\ x\ (\mathit{LAB}_g^{\rightarrow}\ (\lambda l' \to \mathit{JPZ}_g\ l'\ (\mathit{comp}_g\ y\ (\mathit{POP}_g\ (\mathit{JMP}_g\ l))))\ c))
\end{array}
$$

Fig. 8. Graph-based compiler for the while language.

[2017] introduced a high-level combinator library to construct graphs, and the structured graphs of [Oliveira and Cook 2012] make use of higher-order abstract syntax to avoid explicit names.

Reasoning. To the best of our knowledge, there is little work on the use of structured graphs for formal reasoning. Oliveira and Cook [2012] present some equational laws for *map* and *fold* for a simple type of structured graphs, namely cyclic streams. Bahr [2014] uses structured graphs to implement a graph-based compiler and prove it correct in two stages: first, prove a tree-based compiler correct, essentially the correctness property (1) of *comp*; and secondly, prove that the graph-based compiler is equivalent to the tree-based compiler, essentially the correctness property (4) of comp_g. The main differences to the present work is that Bahr proves correctness post hoc, whereas we calculate compilers from their specification, and his work is limited to acyclic graphs and terminating languages, whereas we considered cyclic graphs and non-termination.

Cyclic control-flow. Cyclic graphs mark a qualitative difference in the expressive power of the target language compared to acyclic graphs, as graphs can no longer be represented by finite trees if they contain cycles. Some previous work does consider languages with cyclic control-flow, e.g. Bahr and Hutton [2022] calculate a compiler for a language with while loops. However, as the target language is limited to finite trees, the looping behaviour is instead imitated by repeatedly copying the code for a while loop to the stack and retrieving it later when it has to be executed again. Meijer [1992] also calculates a compiler for a language with while loops, starting with a least fixed-point semantics of such loops. In Meijer's approach, the syntax of the target language is not formally defined but is instead simply made up of the calculated combinators that are used to describe the source language semantics. Depending on the semantics of the meta language, one could interpret the target code as infinite trees. But it is not clear how the least fixed-points of the semantics give rise to a co-inductive syntax for the target language.

Control-flow graphs. The code produced by the graph-based compiler $\mathit{compile}_g$ corresponds to a control-flow graph. In compiler implementations, such graphs typically use concrete labels, i.e. unique identifiers, to specify targets of branching control flow [Ramsey and Dias 2006; Ramsey et al. 2010]. We also find this representation of control-flow using concrete labels in large-scale compiler verification efforts, e.g. CompCert [Leroy 2009] and CakeML [Kumar et al. 2014].

7 Conclusion and Further Work

We have shown how existing work on deriving correct-by-construction compilers that produce tree-structured code can be extended to instead produce graph-structured code. While our method is calculational and hence naturally complements a calculational approach such as Meijer [1992] and Bahr and Hutton [2015], it also readily applies to other methods such as Wand [1995] and

Ager et al. [2003]. By combining our method with existing tree-based calculation methods, we can calculate graph-based compilers for lambda calculi [Bahr and Hutton 2022], concurrent source languages [Bahr and Hutton 2023] and register-based target languages [Bahr and Hutton 2020].

The key issue when calculating graph-based compilers is dealing with names, i.e. labels that denote code pointers. We used parametric higher-order abstract syntax [Chlipala 2008] because it requires little additional machinery, both for pen and paper proofs, and when using a proof assistant. The accompanying formalisation includes an alternative representation for graph-based code using de Bruijn indices [de Bruijn 1972]. It may also be useful to consider other representations for names, particularly if good support for them is available in a proof assistant.

Our graph-based code type $Code_g$ has two labelling constructors $LAB_g^{\rightarrow}$ and $LAB_g^{\leftarrow}$ for forward and for backward jumps, respectively. Some graphs are not expressible using only these constructors, such as those with overlapping jumps, but so far we have not found the need for more general graph constructors in our compiler calculations. However, $LAB_g^{\rightarrow}$ and $LAB_g^{\leftarrow}$ can be combined into a more general constructor $LAB_g :: (Vec\ l\ n \rightarrow Vec\ (Code_g\ l)\ n) \rightarrow Code_g\ l$ similarly to the original work on structured graphs [Oliveira and Cook 2012] to extend expressiveness if needed.

Our use of step-indexing for equational reasoning about coinductive types, namely *Partial* and *Code*, is also motivated by the fact that it requires relatively little formal machinery, and can be readily used in many proof assistants. More advanced methods for coinductive definitions and reasoning, such as guarded recursion [Nakano 2000] and sized types [Hughes et al. 1996], could allow compiler calculations to be streamlined in a proof assistant that supports these features. In particular if combined with cubical type theory [Birkedal et al. 2016], guarded types support powerful equational reasoning principles for coinductive types [Møgelberg and Veltri 2019].

Acknowledgements

We thank the referees for many useful comments and suggestions. This work was partially funded by the Engineering and Physical Sciences Research Council (EPSRC) grant EP/Y010744/1, *Semantics-Directed Compiler Construction: From Formal Semantics to Certified Compilers*.

References

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

Robert Atkey. 2009. Syntax for Free: Representing Syntax with Binding Using Parametricity. In *Typed Lambda Calculi and Applications*. Lecture Notes in Computer Science, Vol. 5608.

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

Patrick Bahr. 2014. Proving Correctness of Compilers Using Structured Graphs. In *Functional and Logic Programming (Lecture Notes in Computer Science, Vol. 8475).*

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

Patrick Bahr and Graham Hutton. 2020. Calculating Correct Compilers II: Return of the Register Machines. *Journal of Functional Programming* 30 (2020).

Patrick Bahr and Graham Hutton. 2022. Monadic Compiler Calculation. *Proceedings of the ACM on Programming Languages* 6, ICFP, Article 93 (2022).

Patrick Bahr and Graham Hutton. 2023. Calculating Compilers for Concurrency. *Proceedings of the ACM on Programming Languages* 7, ICFP, Article 213 (2023).

Patrick Bahr and Graham Hutton. 2024. Supplementary Material for "Beyond Trees: Calculating Graph-Based Compilers". https://doi.org/10.5281/zenodo.11233589

Lars Birkedal, Aleš Bizjak, Ranald Clouston, Hans Bugge Grathwohl, Bas Spitters, and Andrea Vezzosi. 2016. Guarded Cubical Type Theory: Path Equality for Guarded Recursion. Schloss-Dagstuhl - Leibniz Zentrum für Informatik.

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

Venanzio Capretta. 2005. General Recursion via Coinductive Types. *Logical Methods in Computer Science* 1, 2 (2005).

Adam Chlipala. 2008. Parametric Higher-Order Abstract Syntax for Mechanized Semantics. In *Proceedings of the International Conference on Functional Programming.*

Nils Anders Danielsson. 2012. Operational Semantics Using the Partiality Monad. In *Proceedings of the International Conference on Functional Programming*.

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

N.G de Bruijn. 1972. Lambda Calculus Notation with Nameless Dummies, A Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem. *Indagationes Mathematicae* 75, 5 (1972).

Martin Erwig. 2001. Inductive Graphs and Functional Graph Algorithms. *Journal of Functional Programming* 11, 5 (2001).

Jeremy Gibbons. 2021. Continuation-Passing Style, Defunctionalization, Accumulations, and Associativity. *The Art, Science, and Engineering of Programming* 6, 2 (2021).

John Hughes, Lars Pareto, and Amr Sabry. 1996. Proving the Correctness of Reactive Systems Using Sized Types. In *Proceedings of the 23rd Symposium on Principles of Programming Languages*.

Graham Hutton and Patrick Bahr. 2017. Compiling a 50-Year Journey. *Journal of Functional Programming* 27 (2017).

Graham Hutton and Joel Wright. 2004. Compiling Exceptions Correctly. In *Proceedings of the International Conference on Mathematics of Program Construction (Lecture Notes in Computer Science, Vol. 3125)*.

Yugo Kashiwagi and David Wise. 1991. *Graph Algorithms in a Lazy Functional Programming Language*.

David King and John Launchbury. 1995. Structuring Depth-First Search Algorithms in Haskell. In *Proceedings of the 22nd Symposium on Principles of Programming Languages*.

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. *ACM SIGPLAN Notices* 49, 1 (2014).

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

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

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

Andrey Mokhov. 2017. Algebraic Graphs with Class (Functional Pearl). In *Proceedings of the 10th Symposium on Haskell*.

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

Hiroshi Nakano. 2000. A Modality for Recursion. In *Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science*.

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

Bruno Oliveira and William Cook. 2012. Functional Programming with Structured Graphs. In *Proceedings of the International Conference on Functional Programming*.

Mitchell Pickard and Graham Hutton. 2021. Calculating Dependently-Typed Compilers. *Proceedings of the ACM on Programming Languages* 5, ICFP, Article 82 (2021).

Norman Ramsey and João Dias. 2006. An Applicative Control-Flow Graph Based on Huet's Zipper. In *Proceedings of the Workshop on ML*.

Norman Ramsey, João Dias, and Simon Peyton Jones. 2010. Hoopl: A Modular, Reusable Library for Dataflow Analysis and Transformation. In *Proceedings of the Third Symposium on Haskell*.

Mitchell Wand. 1982. Deriving Target Code as a Representation of Continuation Semantics. *ACM Transactions on Programming Languanges and Systems* 4, 3 (1982).

Mitchell Wand. 1995. Compiler Correctness for Parallel Languages. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*.

Received 2024-02-28; accepted 2024-06-18