Proving Correctness of Compilers using Structured Graphs

Patrick Bahr

Department of Computer Science, University of Copenhagen
paba@diku.dk

Abstract. We present an approach to compiler implementation using Oliveira and Cook's structured graphs that avoids the use of explicit jumps in the generated code. The advantage of our method is that it takes the implementation of a compiler using a tree type along with its correctness proof and turns it into a compiler implementation using a graph type along with a correctness proof. The implementation and correctness proof of a compiler using a tree type without explicit jumps is simple, but yields code duplication. Our method provides a convenient way of improving such a compiler without giving up the benefits of simple reasoning.

1 Introduction

Verification of compilers – like other software – is difficult [11]. In such an endeavour one typically has to balance the "cleverness" of the implementation with the simplicity of reasoning about it. A concrete example of this fact is given by Hutton and Wright [9] who present correctness proofs of compilers for a simple language with exceptions. The authors first present a naïve compiler implementation that produces a tree representing the possible control flow of the input program. The code that it produces is essentially the right code, but the compiler *loses information* since it duplicates code instead of sharing it. However, the simplicity of the implementation is matched with a clean and simple proof by equational reasoning. Hutton and Wright also present a more realistic compiler, which uses labels and explicit jumps, resulting in a target code in linear form and without code duplication. However, the cleverer implementation also requires a more complicated proof, in which one has to reason about the freshness and scope of labels.

In this paper we present an intermediate approach, which is still simple, both in its implementation and in its correctness proof, but which avoids the loss of information of the simple approach described by Hutton and Wright [9]. The remedy for the information loss of the simple approach is obvious: we use a graph instead of a tree structure to represent the target code. The linear representation with labels and jumps is essentially a graph as well – it is just a very inconvenient one for reasoning. Instead of using unique names to represent sharing, we use the *structured graphs* representation of Oliveira and Cook [15]. This representation of graphs uses parametric higher-order abstract syntax (PHOAS) [4] to represent binders, which in turn is used to represent sharing. This structure allows us to take the simple compiler implementation using trees, make a slight adjustment to it, and obtain a compiler implementation using graphs that preserves the sharing information.

The key observation that also keeps the correctness proof simple is that the semantics of the two target languages, i.e. their respective *virtual machines*, are equivalent after *unravelling* of the graph structure. More precisely, given the semantics of the tree-based and the graph-based target language as $exec_{\mathsf{T}}$ and $exec_{\mathsf{G}}$, respectively, we have the following equation:

$$exec_{\mathsf{G}} = exec_{\mathsf{T}} \circ unravel$$

We show that this correspondence is an inherent consequence of the recursion schemes that are used to define these semantics. That is, the above property is independent of the object language of the compiler. As a consequence, the correctness proof of the improved, graph-based compiler is reduced to a proof that its implementation is equivalent to the tree-based implementation modulo unravelling. More precisely, it then suffices to show that

$$comp_{\mathsf{T}} = unravel \circ comp_{\mathsf{G}}$$

which is achieved by a straightforward induction proof.

In sum, the technique that we propose here improves existing simple compiler implementations to more realistic ones using a graph representation for the target code. This improvement requires minimal effort – both in terms of the implementation and the correctness proof. The fact that we consider both the implementation and its correctness proof makes our technique the ideal companion to improve a compiler that has been obtained by calculation [14]. Such calculations derive a compiler from a specification, and produce not only an implementation of the compiler but also a proof of its correctness. The example compiler that we use in this paper has in fact been calculated in this way by Bahr and Hutton [2], and we have successfully applied our technique to other compilers derived by Bahr and Hutton [2], which includes compilers for languages with features such as (synchronous and asynchronous) exceptions, (global and local) state and non-determinism. Thus, despite its simplicity, our technique is quite powerful, especially when combined with other techniques such as the abovementioned calculation techniques.

In short, the contributions of this paper are the following:

- From a compiler with code duplication we derive a compiler that avoids duplication using a graph representation.
- We prove that folds over graphs are equal to corresponding folds over the unravelling of the input graphs.
- Using the above result, we derive the correctness of the graph-based compiler implementation from the correctness of the tree based compiler.
- We further simplify the proof by using free monads to represent tree types together with a corresponding monadic graph type.

Throughout this paper we use Haskell [12] as the implementation language. Moreover, the paper is written as a literate Haskell file, which can be compiled using the *Glasgow Haskell Compiler* (GHC). The source file along with the Coq formalisation of the proofs can be found in the associated material[1].

2 A Simple Compiler

The example language that we use throughout the paper is a simple expression language with integers, addition and exceptions:

$$
\begin{aligned}
\mathbf{data}\ \mathit{Expr} &= \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ \ \ \mathit{Expr}\ \mathit{Expr} \\
&\mid\ \mathit{Throw}\ \mid \mathit{Catch}\ \mathit{Expr}\ \mathit{Expr}
\end{aligned}
$$

The semantics of this language is defined using an evaluation function that evaluates a given expression to an integer value or returns *Nothing* in case of an uncaught exception:

$$
\begin{array}{l}
\mathit{eval} :: \mathit{Expr} \rightarrow \mathit{Maybe}\ \mathit{Int} \\
\mathit{eval}\ (\mathit{Val}\ n) \qquad = \mathit{Just}\ n \\
\mathit{eval}\ (\mathit{Add}\ x\ y) \quad = \mathbf{case}\ \mathit{eval}\ x\ \mathbf{of} \\
\qquad\qquad\qquad\quad \mathit{Nothing} \rightarrow \mathit{Nothing} \\
\qquad\qquad\qquad\quad \mathit{Just}\ n \quad \rightarrow \mathbf{case}\ \mathit{eval}\ y\ \mathbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Nothing} \rightarrow \mathit{Nothing} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Just}\ m \quad \rightarrow \mathit{Just}\ (n + m) \\
\mathit{eval}\ \ \mathit{Throw} \qquad = \mathit{Nothing} \\
\mathit{eval}\ (\mathit{Catch}\ x\ h) = \mathbf{case}\ \mathit{eval}\ x\ \mathbf{of} \\
\qquad\qquad\qquad\quad \mathit{Nothing} \rightarrow \mathit{eval}\ h \\
\qquad\qquad\qquad\quad \mathit{Just}\ n \quad \rightarrow \mathit{Just}\ n
\end{array}
$$

This is the same language and semantics used by Hutton and Wright [9]. Like Hutton and Wright, we chose a simple language in order to focus on the essence of the problem, which in our case is control flow in the target language and the use of duplication or sharing to represent it. Moreover, this choice allows us to compare our method to the original work of Hutton and Wright whose focus was on the simplicity of reasoning.

The target for the compiler is a simple stack machine with the following instruction set:

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

The intended semantics (which is made precise later) for the instructions is:

- *PUSH* n pushes an integer value n on the top of the stack,
- *ADD* expects the two topmost stack elements to be integers and replaces them with their sum,

[1] See `http://diku.dk/~paba/graphs.tgz`.

- *MARK* c pushes the code c on the stack, which is meant for handling exceptions,
- *UNMARK* removes such a handler code from the stack,
- *THROW* throws an exception, which causes an unwinding of the stack until

a handler code is reached, and

– *HALT* stops the execution.

For the implementation of the compiler we deviate slightly from the presentation of Hutton and Wright [9] and instead write the compiler in a style that uses additional accumulation parameter c, which simplifies the proofs [8]:

$$
\begin{array}{ll}
\mathit{comp}^{\mathsf{A}} :: \mathit{Expr} \to \mathit{Code} \to \mathit{Code} & \\
\mathit{comp}^{\mathsf{A}}\ (\mathit{Val}\ n) & c = \mathit{PUSH}\ n\ c \\
\mathit{comp}^{\mathsf{A}}\ (\mathit{Add}\ x\ y) & c = \mathit{comp}^{\mathsf{A}}\ x\ (\mathit{comp}^{\mathsf{A}}\ y\ (\mathit{ADD}\ c)) \\
\mathit{comp}^{\mathsf{A}}\ \mathit{Throw} & c = \mathit{THROW} \\
\mathit{comp}^{\mathsf{A}}\ (\mathit{Catch}\ x\ h) & c = \mathit{MARK}\ (\mathit{comp}^{\mathsf{A}}\ h\ c)\ (\mathit{comp}^{\mathsf{A}}\ x\ (\mathit{UNMARK}\ c))
\end{array}
$$

Since the code generator is implemented in this code continuation passing style, function application corresponds to concatenation of code fragments. To stress this reading, we shall use the operator $\triangleright$, which is simply defined as function composition and is declared to associate to the right with minimal precedence:

$$
\begin{array}{l}
(\triangleright) :: (a \to b) \to a \to b \\
f \triangleright x = f\ x
\end{array}
$$

For instance, the equation for the *Add* case of the definition of $\mathit{comp}^{\mathsf{A}}$ then reads:

$$comp^{\mathsf{A}}\ (Add\ x\ y)\ c = comp^{\mathsf{A}}\ x \triangleright comp^{\mathsf{A}}\ y \triangleright ADD \triangleright c$$

To obtain the final code for an expression, we supply $HALT$ as the initial accumulator to $comp^{\mathsf{A}}$. The use of the $\triangleright$ operator to supply the argument indicates the intuition that $HALT$ is placed at the end of the code produced by $comp^{\mathsf{A}}$:

$$\begin{aligned}
&comp :: Expr \rightarrow Code\\
&comp\ e = comp^{\mathsf{A}}\ e \triangleright HALT
\end{aligned}$$

The following examples illustrate the workings of the compiler *comp*:

$$\begin{aligned}
&comp\ (Add\ (Val\ 2)\ (Val\ 3)) &&\rightsquigarrow PUSH\ 2 \triangleright PUSH\ 3 \triangleright ADD \triangleright HALT\\
&comp\ (Catch\ (Val\ 2)\ (Val\ 3)) &&\rightsquigarrow MARK\ (PUSH\ 3 \triangleright HALT)\\
& && \qquad \triangleright PUSH\ 2 \triangleright UNMARK \triangleright HALT\\
&comp\ (Catch\ Throw\ (Val\ 3)) &&\rightsquigarrow MARK\ (PUSH\ 3 \triangleright HALT) \triangleright THROW
\end{aligned}$$

For the virtual machine that executes the code produced by the above compiler, we use the following type for the stack:

$$\begin{aligned}
&\mathbf{type}\ Stack = [Item]\\
&\mathbf{data}\ Item\ \ = VAL\ Int \mid HAN\ (Stack \rightarrow Stack)
\end{aligned}$$

This type deviates slightly from the one for the virtual machine defined by Hutton and Wright [9]. Instead of having the code for the handler on the stack (constructor *HAN*), we have the continuation of the virtual machine on the stack. This will simplify the proof as we shall see later on. However, this type and the accompanying definition of the virtual machine that is given below is exactly the result of the calculation given by Bahr and Hutton [2] just before the last calculation step (which then yields the virtual machine of Hutton and Wright [9]). The virtual machine that works on this stack is defined as follows:

$$
\begin{array}{lll}
\multicolumn{3}{l}{exec :: Code \to Stack \to Stack} \\
exec\ (PUSH\ n\ c) & s = & exec\ c\ (VAL\ n : s) \\
exec\ (ADD\ c) & s = & \textbf{case}\ s\ \textbf{of} \\
 & & \quad (VAL\ m : VAL\ n : s') \to exec\ c\ (VAL\ (n+m) : s') \\
exec\ THROW & s = & unwind\ s \\
exec\ (MARK\ h\ c) & s = & exec\ c\ (HAN\ (exec\ h) : s) \\
exec\ (UNMARK\ c) & s = & \textbf{case}\ s\ \textbf{of} \\
 & & \quad (x : HAN\ \_ : s') \to exec\ c\ (x : s') \\
exec\ HALT & s = & s
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{unwind :: Stack \to Stack} \\
unwind\ [\,] & = \ [\,]
\end{array}
$$

$$\begin{aligned} unwind\ (VAL\ \_ : s) &= unwind\ s \\ unwind\ (HAN\ h : s) &= h\ s \end{aligned}$$

The virtual machine does what is expected from the informal semantics that we have given above. The semantics of *MARK*, however, may seem counterintuitive at first: *MARK* does not put the handler code on the stack but rather the continuation that is obtained by executing it. Consequently, when the unwinding of the stack reaches a handler h on the stack, this handler h is directly applied to the remainder of the stack. This slight deviation from the semantics of Hutton and Wright [9] makes sure that *exec* is in fact a fold.

We will not go into the details of the correctness proof for the compiler *comp*. One can show that it satisfies the following correctness property [2]:

Theorem 1 (compiler correctness).

$$exec\ (comp\ e)\ [\,] = conv\ (eval\ e)$$

$$where \qquad \begin{aligned} conv\ (Just\ n) &= [\,Val\ n\,] \\ conv\ Nothing &= [\,] \end{aligned}$$

That is, in particular, we have that

$$exec\ (comp\ e)\ [\,] = [\,Val\ n\,] \quad \Longleftrightarrow \quad eval\ e = Just\ n$$

While the compiler has the nice property that it can be derived from the language semantics, the code that it produces is quite unrealistic. Note the duplication that occurs for generating the code for *Catch*: the continuation code c

is inserted both after the handler code (in $comp^{\mathsf{A}}\ h\ c$) and after the *UNMARK* instruction. This is necessary since the code c should be executed regardless whether an exception is thrown or not.

This duplication can be avoided by using explicit jumps in the code. Instead of duplicating code, jumps to a single copy of the code are inserted. However, this complicates both the implementation of the compiler and its correctness proof [9]. Also the derivation of such a compiler is by calculation is equally cumbersome.

The approach that we suggest in this paper derives a slightly different compiler that instead of a tree structure produces a graph structure. Along with the compiler we derive a virtual machine that also works on the graph structure. The two variants of the compiler and its companion virtual machine only differ in the sharing that the graph variant provides. This fact allows us to derive the correctness of the graph-based compiler very easily from the correctness of the original tree-based compiler. In particular, we use the *structured graphs* of Oliveira and Cook [15], which will allow us to implement the compiler without code duplication and prove its correctness with little overhead.

3 From Trees to Graphs

Before we derive the graph-based compiler and the corresponding virtual machine, we restructure the definition of the original compiler and the corresponding virtual machine. This will smoothen the process and simplify the presentation.

3.1 Preparations

Instead of defining the type *Code* directly, we represent it as the initial algebra of a functor. To distinguish this representation from the later graph representation, we use the name *Tree* for the initial algebra construction.

$$\mathbf{data}\ \mathit{Tree}\ f = \mathit{In}\ (f\ (\mathit{Tree}\ f))$$

The functor that induces the initial algebra that we shall use for representing the target language is defined as follows:

$$
\begin{array}{llll}
\textbf{data}\ \mathit{Code}\ a = \mathit{PUSH}\ \mathit{Int}\ a & \mid \mathit{ADD}\ a & \mid \mathit{HALT} \\
\qquad\qquad\qquad\quad \mid \mathit{MARK}\ a\ a & \mid \mathit{UNMARK}\ a & \mid \mathit{THROW}
\end{array}
$$

The type representing the target code is thus *Tree Code*. We proceed by reformulating the definition of *comp* to work on the type *Tree Code*:

$$
\begin{array}{ll}
\mathit{comp}^{\mathsf{A}}_{\mathsf{T}} :: \mathit{Expr} \to \mathit{Tree}\ \mathit{Code} \to \mathit{Tree}\ \mathit{Code} & \\
\mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ (\mathit{Val}\ n) & c = \mathit{PUSH}_{\mathsf{T}}\ n \triangleright c \\
\mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ (\mathit{Add}\ x\ y) & c = \mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ x \triangleright \mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ y \triangleright \mathit{ADD}_{\mathsf{T}} \triangleright c \\
\mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ \ \mathit{Throw} & c = \mathit{THROW}_{\mathsf{T}} \\
\mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ (\mathit{Catch}\ x\ h) & c = \mathit{MARK}_{\mathsf{T}}\ (\mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ h \triangleright c) \triangleright \mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ x \triangleright \mathit{UNMARK}_{\mathsf{T}} \triangleright c
\end{array}
$$

$$
\begin{array}{l}
\mathit{comp}_{\mathsf{T}} :: \mathit{Expr} \to \mathit{Tree}\ \mathit{Code} \\
\mathit{comp}_{\mathsf{T}}\ e = \mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ e \triangleright \mathit{HALT}_{\mathsf{T}}
\end{array}
$$

Note that we do not use the constructors of *Code* directly, but instead we use *smart constructors* that also apply the constructor *In* of the type constructor *Tree*. For example, $\mathit{PUSH}_{\mathsf{T}}$ is defined as follows:

$$
\mathit{PUSH}_{\mathsf{T}} :: \mathit{Int} \to \mathit{Tree}\ \mathit{Code} \to \mathit{Tree}\ \mathit{Code}
$$

$$PUSH_{\mathsf{T}}\ i\ c = In\ (PUSH\ i\ c)$$

Lastly, we also reformulate the semantics of the target language, i.e. we define the function *exec* on the type *Tree Code*. To do this, we use the following definition of a fold on an initial algebra:

$$\begin{array}{l}
fold :: Functor\ f \Rightarrow (f\ r \rightarrow r) \rightarrow Tree\ f \rightarrow r \\
fold\ alg\ (In\ t) = alg\ (fmap\ (fold\ alg)\ t)
\end{array}$$

The definition of the semantics is a straightforward transcription of the definition of *exec* into an algebra:

$$\begin{array}{ll}
\multicolumn{2}{l}{execAlg :: Code\ (Stack \rightarrow Stack) \rightarrow Stack \rightarrow Stack} \\
execAlg\ (PUSH\ n\ c) & s = c\ (VAL\ n : s) \\
execAlg\ (ADD\ c) & s = \textbf{case}\ s\ \textbf{of} \\
 & \qquad (VAL\ m : VAL\ n : s') \rightarrow c\ (VAL\ (n+m) : s') \\
execAlg\ THROW & s = unwind\ s \\
execAlg\ (MARK\ h\ c) & s = c\ (HAN\ h : s) \\
execAlg\ (UNMARK\ c) & s = \textbf{case}\ s\ \textbf{of} \\
 & \qquad (x : HAN\ _ : s') \rightarrow c\ (x : s') \\
execAlg\ HALT & s = s
\end{array}$$

$$exec_{\mathsf{T}} :: Tree\ Code \rightarrow Stack \rightarrow Stack$$

$$exec_{\top} = \mathit{fold}\ \mathit{execAlg}$$

From the correctness of the original compiler from Section 2, as expressed in Theorem 1, we obtain the correctness of our reformulation of the implementation:

Corollary 1 (correctness of $comp_{\top}$).

$$exec_{\top}\ (comp_{\top}\ e)\ [\,] = conv\ (eval\ e)$$

3.2 Deriving a Graph-Based Compiler

Finally, we turn to the graph-based implementation of the compiler. Essentially, this implementation is obtained from $comp_{\top}$ by replacing the type *Tree Code* with a type *Graph Code*, which instead of a tree structure has a graph structure, and using explicit sharing instead of duplication.

In order to define graphs over a functor, we use the representation of Oliveira and Cook [15] called *structured graphs*. Put simply, a structured graph is a tree

with added sharing facilitated by let bindings. In turn, let bindings are represented using parametric higher-order abstract syntax [4].

$$\begin{aligned}\textbf{data}\ \mathit{Graph}'\ f\ v = {} & \mathit{GIn}\ (f\ (\mathit{Graph}'\ f\ v)) \\ \mid {} & \mathit{Let}\ (\mathit{Graph}'\ f\ v)\ (v \to \mathit{Graph}'\ f\ v) \\ \mid {} & \mathit{Var}\ v\end{aligned}$$

The first constructor has the same structure has the constructor of the *Tree* type constructor. The other two constructors will allow us to express let bindings: $\mathit{Let}\ g\ (\lambda x \to h)$ binds g to the metavariable x in h. Metavariables bound in a let binding have type v; the only way to use them is with the constructor *Var*. To enforce this invariant, the type variable v is made polymorphic:

$$\textbf{newtype}\ \mathit{Graph}\ f = \mathit{Graph}\ \{\mathit{unGraph} :: \forall\ v\ .\ \mathit{Graph}'\ f\ v\}$$

We shall use the type constructor *Graph* (and *Graph′*) as a replacement for *Tree*. For the purposes of our compiler we only need acyclic graphs. That is why we only consider non-recursive let bindings as opposed to the more general structured graphs of Oliveira and Cook [15]. This restriction to non-recursive let bindings is crucial for the reasoning principle that we use to prove correctness.

We can use the graph type almost as a drop-in replacement for the tree type. The only thing that we need to do is to use smart constructors that use the constructor *GIn* instead of *In*, e.g.

$$
\begin{array}{l}
\mathit{PUSH}_{\mathsf{G}} :: \mathit{Int} \to \mathit{Graph}'\ \mathit{Code}\ v \to \mathit{Graph}'\ \mathit{Code}\ v \\
\mathit{PUSH}_{\mathsf{G}}\ i\ c = \mathit{GIn}\ (\mathit{PUSH}\ i\ c)
\end{array}
$$

From the type of the smart constructors we can observe that graphs are constructed using the type constructor *Graph′*, not *Graph*. Only after the construction of the graph is completed, the constructor *Graph* is applied in order to obtain a graph of type *Graph Code*.

The definition of $\mathit{comp}^{\mathsf{A}}_{\mathsf{T}}$ can be transcribed into graph style by simply using the abovementioned smart constructors instead:

$$
\begin{array}{lll}
\mathit{comp}^{\mathsf{A}}_{\mathsf{G}} :: \mathit{Expr} \to \mathit{Graph}'\ \mathit{Code}\ a \to \mathit{Graph}'\ \mathit{Code}\ a \\
\mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ (\mathit{Val}\ n) & c = \mathit{PUSH}_{\mathsf{G}}\ n \rhd c \\
\mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ (\mathit{Add}\ x\ y) & c = \mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ x \rhd \mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ y \rhd \mathit{ADD}_{\mathsf{G}} \rhd c \\
\mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ (\mathit{Throw}) & c = \mathit{THROW}_{\mathsf{G}} \\
\mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ (\mathit{Catch}\ x\ h) & c = \mathit{MARK}_{\mathsf{G}}\ (\mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ h \rhd c) \rhd \mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ x \rhd \mathit{UNMARK}_{\mathsf{G}} \rhd c
\end{array}
$$

The above is a one-to-one transcription of $\mathit{comp}^{\mathsf{A}}_{\mathsf{T}}$. But this is not what we want. We want to make use of the fact that the target language allows sharing. In particular, we want to get rid of the duplication in the code generated for *Catch*.

We can avoid this duplication by simply using a let binding to replace the two occurrences of c with a metavariable c' that is then bound to c. The last equation for $comp^{\mathsf{A}}_{\mathsf{G}}$ is thus rewritten as follows:

$$\begin{aligned}
comp^{\mathsf{A}}_{\mathsf{G}}\ (Catch\ x\ h)\ c = Let\ c\ (\lambda c' \to MARK_{\mathsf{G}}\ (comp^{\mathsf{A}}_{\mathsf{G}}\ h \triangleright Var\ c') \\
\triangleright comp^{\mathsf{A}}_{\mathsf{G}}\ x \triangleright UNMARK_{\mathsf{G}} \triangleright Var\ c')
\end{aligned}$$

The right-hand side for the case $Catch\ x\ h$ has now only one occurrence of c.

The final code generator function $comp^{\mathsf{A}}_{\mathsf{G}}$ is then obtained by supplying $HALT_{\mathsf{G}}$ as the initial value of the code continuation and wrapping the result so as to return a result of type *Graph Code*:

$$\begin{aligned}
&comp_{\mathsf{G}} :: Expr \to Graph\ Code \\
&comp_{\mathsf{G}}\ e = Graph\ (comp^{\mathsf{A}}_{\mathsf{G}}\ e \triangleright HALT_{\mathsf{G}})
\end{aligned}$$

This definition makes explicit that the result type of $comp^{\mathsf{A}}_{\mathsf{G}}$ is parametric in the type v of metavariables. This parametricity makes sure that the graphs we get are in fact well-defined.

To illustrate the difference between $comp_{\mathsf{G}}$ and $comp_{\mathsf{T}}$, we apply both of them to an example expression $e = Add\ (Catch\ (Val\ 1)\ (Val\ 2))\ (Val\ 3)$:

$$\begin{aligned}
comp_{\mathsf{T}}\ e \leadsto MARK\ (PUSH_{\mathsf{T}}\ 2 \triangleright PUSH_{\mathsf{T}}\ 3 \triangleright ADD_{\mathsf{T}} \triangleright HALT_{\mathsf{T}}) \\
\triangleright PUSH_{\mathsf{T}}\ 1 \triangleright UNMARK_{\mathsf{T}} \triangleright PUSH_{\mathsf{T}}\ 3 \triangleright ADD_{\mathsf{T}} \triangleright HALT_{\mathsf{T}}
\end{aligned}$$

$$
\begin{aligned}
&\mathit{comp}_{\mathsf{G}}\ e \rightsquigarrow \mathit{Let}\ (\mathit{PUSH}_{\mathsf{G}}\ 3 \triangleright \mathit{ADD}_{\mathsf{G}} \triangleright \mathit{HALT}_{\mathsf{G}})\ (\lambda v \rightarrow \\
&\qquad \mathit{MARK}_{\mathsf{G}}\ (\mathit{PUSH}_{\mathsf{G}}\ 2 \triangleright \mathit{Var}\ v) \triangleright \mathit{PUSH}_{\mathsf{G}}\ 1 \triangleright \mathit{UNMARK}_{\mathsf{G}} \triangleright \mathit{Var}\ v)
\end{aligned}
$$

Note that $\mathit{comp}_{\mathsf{T}}$ duplicates the code fragment $\mathit{PUSH}_{\mathsf{T}}\ 3 \triangleright \mathit{ADD}_{\mathsf{T}} \triangleright \mathit{HALT}_{\mathsf{T}}$, which is supposed to be executed after the catch expression, whereas $\mathit{comp}_{\mathsf{G}}$ binds this code fragment to a metavariable v, which is then used as a substitute.

The recursion schemes on structured graphs make use of the parametricity in the variable type as well. The general fold over graphs as given by Oliveira and Cook [15] is defined as follows:[2]

$$
\begin{aligned}
&\mathit{gfold} :: \mathit{Functor}\ f \Rightarrow (v \rightarrow r) \rightarrow (r \rightarrow (v \rightarrow r) \rightarrow r) \rightarrow (f\ r \rightarrow r) \\
&\quad \rightarrow \mathit{Graph}\ f \rightarrow r \\
&\mathit{gfold}\ v\ l\ i\ (\mathit{Graph}\ g) = \mathit{trans}\ g\ \textbf{where} \\
&\quad \mathit{trans}\ (\mathit{Var}\ x) \quad = v\ x \\
&\quad \mathit{trans}\ (\mathit{Let}\ e\ f) = l\ (\mathit{trans}\ e)\ (\mathit{trans} \circ f) \\
&\quad \mathit{trans}\ (\mathit{GIn}\ t) \quad = i\ (\mathit{fmap}\ \mathit{trans}\ t)
\end{aligned}
$$

It takes three functions, which are used to interpret the three constructors of *Graph′*. This general form is needed for example if we want to transform the graph representation into a linearised form (see associated material), but for our purposes we only need a simple special case of it:

$$
\begin{aligned}
&\mathit{ufold} :: \mathit{Functor}\ f \Rightarrow (f\ r \to r) \to \mathit{Graph}\ f \to r \\
&\mathit{ufold} = \mathit{gfold}\ \mathit{id}\ (\lambda e\ f \to f\ e)
\end{aligned}
$$

Note that the type signature is identical to the one for *fold* except for the use of *Graph* instead of *Tree*. And indeed the semantics of the two folds are related:

[2] Oliveira and Cook [15] considered the more general case of cyclic graphs, the definition of *gfold* given here is specialised to the case of acyclic graphs.

$\mathit{ufold}\ r\ a\ g$ is equal to $\mathit{fold}\ r\ a\ t$, where t is the *unravelling* of g. This is one of the key properties that we shall use for deriving the correctness theorem for $\mathit{comp}_{\mathsf{G}}$. Moreover, this property allows us to define the semantics of the target language *Graph Code* by reusing the algebra *execAlg* that we defined in Section 3.1 to define the semantics of *Tree Code*:

$$
\begin{aligned}
&\mathit{exec}_{\mathsf{G}} :: \mathit{Graph}\ \mathit{Code} \to \mathit{Stack} \to \mathit{Stack} \\
&\mathit{exec}_{\mathsf{G}} = \mathit{ufold}\ \mathit{execAlg}
\end{aligned}
$$

4 Correctness Proof

In this section we shall prove that the compiler that we defined in Section 3 is indeed correct. This turns to be rather simple: we derive the correctness property for $comp_{\mathsf{G}}$ from the correctness property for $comp_{\mathsf{T}}$. The simplicity of the argument is rooted in the fact that $comp_{\mathsf{T}}$ is the same as $comp_{\mathsf{G}}$ followed by unravelling. In other words, $comp_{\mathsf{G}}$ only differs from $comp_{\mathsf{T}}$ in that it adds sharing – as expected.

4.1 Compiler Correctness by Unravelling

Before we prove this relation between $comp_{\mathsf{T}}$ and $comp_{\mathsf{G}}$, we need to specify what unravelling means:

$$
\begin{aligned}
&\mathit{unravel} :: \mathit{Functor}\ f \Rightarrow \mathit{Graph}\ f \rightarrow \mathit{Tree}\ f \\
&\mathit{unravel} = \mathit{ufold}\ \mathit{In}
\end{aligned}
$$

While this definition is nice and compact, we gain more insight into what it actually does by unfolding it:

$$
\begin{aligned}
&\mathit{unravel} :: \mathit{Functor}\ f \Rightarrow \mathit{Graph}\ f \rightarrow \mathit{Tree}\ f \\
&\mathit{unravel}\ (\mathit{Graph}\ g) = \mathit{unravel}'\ g
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{unravel}' :: \mathit{Functor}\ f \Rightarrow \mathit{Graph}'\ f\ (\mathit{Tree}\ f) \rightarrow \mathit{Tree}\ f \\
&\mathit{unravel}'\ (\mathit{Var}\ x) \quad = x \\
&\mathit{unravel}'\ (\mathit{Let}\ e\ f) = \mathit{unravel}'\ (f\ (\mathit{unravel}'\ e)) \\
&\mathit{unravel}'\ (\mathit{GIn}\ t) \quad = \mathit{In}\ (\mathit{fmap}\ \mathit{unravel}'\ t)
\end{aligned}
$$

We can see that *unravel* simply replaces GIn with In, and applies the function argument f of a let binding to the bound value e. For example, we have that

$$
\begin{aligned}
&\mathit{unravel}\ (\mathit{Graph}\ (\mathit{Let}\ (\mathit{PUSH}_{\mathsf{G}}\ 2)\ (\lambda x \rightarrow \mathit{MARK}_{\mathsf{G}}\ (\mathit{Var}\ x) \triangleright \mathit{Var}\ x))) \\
&\rightsquigarrow \mathit{MARK}_{\mathsf{T}}\ (\mathit{PUSH}_{\mathsf{T}}\ 2) \triangleright \mathit{PUSH}_{\mathsf{T}}\ 2
\end{aligned}
$$

We can now formulate the relation between $\mathit{comp}^{\mathsf{A}}_{\mathsf{T}}$ and $\mathit{comp}^{\mathsf{A}}_{\mathsf{G}}$:

Lemma 1.

$$\mathit{comp}_{\mathsf{T}} = \mathit{unravel} \circ \mathit{comp}_{\mathsf{G}}$$

This lemma, which we shall prove at the end of this section, is one half of the argument for deriving the correctness property for $\mathit{comp}_{\mathsf{G}}$. The other half is the property that $\mathit{exec}_{\mathsf{T}}$ and $\mathit{exec}_{\mathsf{G}}$ have the converse relationship, viz.

$$exec_{\mathsf{G}} = exec_{\mathsf{T}} \circ unravel$$

Proving this property is much simpler, though, because it follows from a more general property of *ufold* and *fold*.

Theorem 2. *Given a strictly positive functor f, a type c, and $alg :: f\ c \to c$, we have the following:*

$$ufold\ alg = fold\ alg \circ unravel$$

The equality $exec_{\mathsf{G}} = exec_{\mathsf{T}} \circ unravel$ is an instance of Theorem 2 where $alg = execAlg$. We defer discussion of the proof of this theorem until Section 4.2.

We can now derive the correctness property of $comp_{\mathsf{G}}$ by combining Lemma 1 and Theorem 2:

Theorem 3 (correctness of $comp_{\mathsf{G}}$).

$$exec_{\mathsf{G}}\ (comp_{\mathsf{G}}\ e)\ [\,] = conv\ (eval\ e) \qquad \textit{for all } e :: Expr$$

Proof.

$$\begin{aligned} exec_{\mathsf{G}}\ (comp_{\mathsf{G}}\ e)\ [\,] &\overset{\text{Thm. 2}}{=} exec_{\mathsf{T}}\ (unravel\ (comp_{\mathsf{G}}\ e)\ [\,] \\ &\overset{\text{Lem. 1}}{=} exec_{\mathsf{T}}\ (comp_{\mathsf{T}}\ e)\ [\,] \end{aligned}$$

$$\overset{\text{Cor. 2}}{=} \mathit{conv}\ (\mathit{eval}\ e) \qquad \square$$

We conclude this section by giving the proof of Lemma 1.

Proof (of Lemma 1). Instead of proving the equation directly, we prove the following equation:

$$\mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ e \triangleright \mathit{unravel}'\ c = \mathit{unravel}'\ (\mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ e \triangleright c) \quad \text{for all } c :: \forall\, v. \mathit{Graph}'\ \mathit{Code}\ v \quad (1)$$

The lemma follows from the above equation as follows:

$$
\begin{aligned}
& \mathit{comp}_{\mathsf{T}}\ e \\
= \quad & \{ \text{ definition of } \mathit{comp}_{\mathsf{T}} \ \} \\
& \mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ e \triangleright \mathit{HALT}_{\mathsf{T}} \\
= \quad & \{ \text{ definition of } \mathit{unravel}' \ \} \\
& \mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ e \triangleright \mathit{unravel}'\ \mathit{HALT}_{\mathsf{G}} \\
= \quad & \{ \text{ Equation (1) } \} \\
& \mathit{unravel}'\ (\mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ e \triangleright \mathit{HALT}_{\mathsf{G}}) \\
= \quad & \{ \text{ definition of } \mathit{unravel} \ \} \\
& \mathit{unravel}\ (\mathit{Graph}\ (\mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ e \triangleright \mathit{HALT}_{\mathsf{G}})) \\
= \quad & \{ \text{ definition of } \mathit{comp}_{\mathsf{G}} \ \}
\end{aligned}
$$

$$unravel\ (comp_{\mathsf{G}}\ e)$$

We prove (1) by induction on e:

– <u>Case $e = Val\ n$:</u>

$$\begin{aligned}
& unravel'\ (comp^{\mathsf{A}}_{\mathsf{G}}\ (Val\ n) \triangleright c) \\
= \quad & \{\ \text{definition of } comp^{\mathsf{A}}_{\mathsf{G}}\ \} \\
& unravel'\ (PUSH_{\mathsf{G}}\ n \triangleright c) \\
= \quad & \{\ \text{definition of } unravel'\ \} \\
& PUSH_{\mathsf{T}}\ n \triangleright unravel'\ c \\
= \quad & \{\ \text{definition of } comp^{\mathsf{A}}_{\mathsf{T}}\ \} \\
& comp^{\mathsf{A}}_{\mathsf{T}}\ (Val\ n) \triangleright unravel'\ c
\end{aligned}$$

– <u>Case $e = Throw$:</u>

$$\begin{aligned}
& unravel'\ (comp^{\mathsf{A}}_{\mathsf{G}}\ Throw \triangleright c) \\
= \quad & \{\ \text{definition of } comp^{\mathsf{A}}_{\mathsf{G}}\ \} \\
& unravel'\ THROW_{\mathsf{G}} \\
= \quad & \{\ \text{definition of } unravel'\ \} \\
& THROW_{\mathsf{T}} \\
= \quad & \{\ \text{definition of } comp^{\mathsf{A}}_{\mathsf{T}}\ \} \\
& comp^{\mathsf{A}}_{\mathsf{T}}\ Throw \triangleright unravel'\ c
\end{aligned}$$

– <u>Case $e = Add\ x\ y$:</u>

$$\begin{aligned}
& unravel'\ (comp^{\mathsf{A}}_{\mathsf{G}}\ (Add\ x\ y) \triangleright c) \\
= \quad & \{\ \text{definition of } comp^{\mathsf{A}}_{\mathsf{G}}\ \} \\
& unravel'\ (comp^{\mathsf{A}}_{\mathsf{G}}\ x \triangleright comp^{\mathsf{A}}_{\mathsf{G}}\ y \triangleright ADD_{\mathsf{G}} \triangleright c) \\
= \quad & \{\ \text{induction hypothesis}\ \}
\end{aligned}$$

$$
\begin{aligned}
& \mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ x \triangleright \mathit{unravel}'\ (\mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ y \triangleright \mathit{ADD}_{\mathsf{G}} \triangleright c) \\
= \quad & \{\ \text{induction hypothesis}\ \} \\
& \mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ x \triangleright \mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ y \triangleright \mathit{unravel}'\ (\mathit{ADD}_{\mathsf{G}} \triangleright c) \\
= \quad & \{\ \text{definition of } \mathit{unravel}'\ \} \\
& \mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ x \triangleright \mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ y \triangleright \mathit{ADD}_{\mathsf{T}} \triangleright \mathit{unravel}'\ c \\
= \quad & \{\ \text{definition of } \mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ \} \\
& \mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ (\mathit{Add}\ x\ y) \triangleright \mathit{unravel}'\ c
\end{aligned}
$$

– Case $e = \mathit{Catch}\ x\ h$:

$$
\begin{aligned}
& \mathit{unravel}'\ (\mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ (\mathit{Catch}\ x\ h) \triangleright c) \\
= \quad & \{\ \text{definition of } \mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ \} \\
& \mathit{unravel}'\ (\mathit{Let}\ c\ (\lambda c' \to \mathit{MARK}_{\mathsf{G}}\ (\mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ h \triangleright \mathit{Var}\ c') \\
& \qquad\qquad\qquad\qquad \triangleright \mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ x \triangleright \mathit{UNMARK}_{\mathsf{G}} \triangleright \mathit{Var}\ c')) \\
= \quad & \{\ \text{definition of } \mathit{unravel}' \text{ and } \beta\text{-reduction}\ \} \\
& \mathit{unravel}'\ (\mathit{MARK}_{\mathsf{G}}\ (\mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ h \triangleright \mathit{Var}\ (\mathit{unravel}'\ c)) \\
& \qquad\qquad \triangleright \mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ x \triangleright \mathit{UNMARK}_{\mathsf{G}} \triangleright \mathit{Var}\ (\mathit{unravel}'\ c)) \\
= \quad & \{\ \text{definition of } \mathit{unravel}'\ \} \\
& \mathit{MARK}_{\mathsf{T}}\ (\mathit{unravel}'\ (\mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ h \triangleright \mathit{Var}\ (\mathit{unravel}'\ c))) \\
& \quad \triangleright \mathit{unravel}'\ (\mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ x \triangleright \mathit{UNMARK}_{\mathsf{G}} \triangleright \mathit{Var}\ (\mathit{unravel}'\ c)) \\
= \quad & \{\ \text{induction hypothesis}\ \}
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{MARK}_{\mathsf{T}}\ (\mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ h \rhd \mathit{unravel}'\ (\mathit{Var}\ (\mathit{unravel}'\ c))) \\
& \quad \rhd \mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ x \rhd \mathit{unravel}'\ (\mathit{UNMARK}_{\mathsf{G}} \rhd \mathit{Var}\ (\mathit{unravel}'\ c)) \\
= & \quad \{\ \text{definition of } \mathit{unravel}'\ \} \\
& \mathit{MARK}_{\mathsf{T}}\ (\mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ h \rhd \mathit{unravel}'\ c) \rhd \mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ x \rhd \mathit{UNMARK}_{\mathsf{T}} \rhd \mathit{unravel}'\ c \\
= & \quad \{\ \text{definition of } \mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ \} \\
& \mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ (\mathit{Catch}\ x\ h) \rhd \mathit{unravel}'\ c
\end{aligned}
$$

□

4.2 Proof of Theorem 2

To conclude this section, we discuss Theorem 2 and its proof. The statement of said theorem is quite intuitive: given a structured graph $g :: \mathit{Graph}\ f$ over a strictly positive functor f, a fold with algebra alg yields the same result as first unravelling g and then folding the resulting tree with alg, i.e.

$$\mathit{ufold}\ \mathit{alg} = \mathit{fold}\ \mathit{alg} \circ \mathit{unravel}$$

When looking at the definition of *unravel* and *ufold*, it becomes intuitively clear why this equality holds: *unravel* inlines let-bindings while *ufold* folds a let binding by folding the bound expression and then inserting the result for each occurrence of the bound metavariable. However, proving this property formally turns out to be quite difficult.

The problem is that both sides of the equation involve a fold over a structured graph and each of the two folds maintain different invariants. Implicitly, every fold over a graph maintains an invariant about metavariables, i.e. subterms of the form $\mathit{Var}\ x$. This invariant is established by which kind of arguments are passed to the function f that is the second argument of the *Let* constructor. For example, the invariant for *unravel* is that, in every occurrence of $\mathit{Var}\ x$, x is the result of $\mathit{unravel}'$ applied to some graph.

As a consequence, for an equality as the one stated in Theorem 2, the two invariants get out of sync when trying to conduct an induction proof.

To avoid this problem, we have reformulated the implementation of structured graphs such that it uses de Bruijn indices for encoding binders instead of PHOAS. Moreover, we have used the technique proposed by Bernardy and Pouillard [3] to provide a PHOAS interface to this implementation of structured graphs. This allows us to use essentially the same simple definition of the graph-based compiler as presented in Section 3.2. Using this representation of structured graphs – PHOAS interface on the outside, de Bruijn indices under the hood – we proved Theorem 2 as well as Lemma 1 in the Coq theorem prover (see associated material).

5 Concluding Remarks

5.1 A Monadic Approach

The proof technique presented in this paper can be refined further by replacing the tree type $\mathit{Tree}\ f$ of a functor f by the free monad type $\mathit{Tree}_{\mathsf{M}}\ f$ of f. The type constructor $\mathit{Tree}_{\mathsf{M}}$ is obtained from Tree by adding a constructor of type $a \to \mathit{Tree}_{\mathsf{M}}\ f\ a$. Likewise, the graph type $\mathit{Graph}\ f$ can be given a monadic structure.

This monadic structure allows us to use the monadic bind operator $\gg$ instead of function application (for which we used the $\triangleright$ notation). That is, we write $\mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ h \gg c$ instead of $\mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ h \triangleright c$ for example. The use of an accumulating parameter in the original compiler implementations is simulated by the monadic

structure (an idea used by Matsuda et al. [13]). As a result, the proof of Lemma 1 can be simplified. Instead of the equation

$$comp^{\mathsf{A}}_{\mathsf{T}}\ e \circ unravel' = unravel' \circ comp^{\mathsf{A}}_{\mathsf{G}}\ e$$

we only have to prove the simpler equation

$$comp^{\mathsf{A}}_{\mathsf{T}} = unravel' \circ comp^{\mathsf{A}}_{\mathsf{G}}$$

This simplifies the induction proof. While this proof requires an additional lemma, viz. that unravelling distributes over $\gg$, this lemma can be proved (once and for all) for any strictly positive functor f:

$$unravel'\ (g_1 \gg g_2) = unravel'\ g_1 \gg unravel'\ g_2$$

The details can be found in the associated material.

5.2 Related and Future Work

Compiler verification is still a hard problem and in this paper we only cover one – but arguably the central – part of a compiler, viz. the translation of a high-level language to a low-level language. The literature on the topic of compiler verification is vast (e.g. see the survey of Dave [6]). More recent work has shown impressive results in verification of a realistic compiler for the C language [11]. But there are also efforts in verifying compilers for more higher-level languages (e.g. by Chlipala [5]).

This paper, however, focuses on identifying simple but powerful techniques for reasoning about compilers rather than engineering massive proofs for full-scale compilers. Our contributions thus follow the work on calculating compilers [19, 14, 1] as well as Hutton and Wright's work on equational reasoning about compilers [9, 10].

Structured graphs have been used in the setting of programming language implementation before: Oliveira and Löh [16] used structured graphs to represent embedded domain-specific languages (EDSLs). That is, graphs are used for the representation of the source language. Graph structures used for representing intermediate languages in a compiler typically employ pointers (e.g. Ramsey and Dias [17]) or labels (e.g. Ramsey et al. [18]). We are not aware of any work that makes use of higher-order abstract syntax or de Bruijn indices in the representation of graph structures in this setting.

The use of structured graphs simplifies both the implementation – there is hardly any syntactic overhead compared to the tree-based implementation – and the reasoning. However, reasoning directly over different folds on such graphs is still a problem as we have described in Section 4.2. Oliveira and Cook [15] present some algebraic laws for reasoning over structured graphs directly, but these laws are restricted to particular instances like cyclic streams and cyclic binary trees.

A shortcoming of our method is its limitation to acyclic graphs. Nevertheless, the implementation part of our method easily generalises to cyclic structures, which permits compilation of cyclic control structures like loops. Corresponding correctness proofs, however, need a different reasoning principle.

Acknowledgements

The author would like to thank Nicolas Pouillard and Daniel Gustafsson for their assistance in the accompanying Coq development.

References

[1] M. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. From interpreter to compiler and virtual machine: A functional derivation. Technical Report RS-03-14, Department of Computer Science, University of Aarhus, 2003.

[2] P. Bahr and G. Hutton. Calculating correct compilers. Unpublished manuscript, 2013.

[3] J.-P. Bernardy and N. Pouillard. Names for free: polymorphic views of names and binders. In *Haskell '13*, pages 13–24, 2013.

[4] A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *ICFP '08*, pages 143–156, 2008.

[5] A. Chlipala. A verified compiler for an impure functional language. In

POPL '10, pages 93–106, 2010.

[6] M. Dave. Compiler verification: a bibliography. *SIGSOFT Softw. Eng. Notes*, 28(6):2–2, 2003.

[7] J. Engelfriet and H. Vogler. Macro tree transducers. *J. Comput. System Sci.*, 31(1):71 – 146, 1985.

[8] G. Hutton. *Programming in Haskell*, volume 2. CUP Cambridge, 2007.

[9] G. Hutton and J. Wright. Compiling exceptions correctly. In D. Kozen, editor, *Mathematics of Program Construction*, volume 3125 of *LNCS*, pages 211–227, 2004.

[10] G. Hutton and J. Wright. What is the meaning of these constant interruptions? *J. Funct. Program.*, 17(06):777–792, 2007.

[11] X. L. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL '06*, pages 42–54, 2006.

[12] S. Marlow. Haskell 2010 language report, 2010.

[13] K. Matsuda, K. Inaba, and K. Nakano. Polynomial-time inverse computation for accumulative functions with multiple data traversals. In *PEPM '12*, pages 5–14, 2012.

[14] E. Meijer. *Calculating Compilers*. PhD thesis, Katholieke Universiteit Nijmegen, 1992.

[15] B. Oliveira and W. Cook. Functional programming with structured graphs. In *ICFP '12*, pages 77–88, 2012.

[16] B. Oliveira and A. Löh. Abstract syntax graphs for domain specific lan-

guages. In *PEPM '13*, pages 87–96, 2013.

[17] N. Ramsey and J. Dias. An applicative control-flow graph based on huet's zipper. In *ML '05*, volume 148, pages 105 – 126, 2006.

[18] N. Ramsey, J. Dias, and S. Peyton Jones. Hoopl: a modular, reusable library for dataflow analysis and transformation. In *Haskell '10*, pages 121–134, 2010.

[19] M. Wand. Deriving target code as a representation of continuation semantics. *ACM Trans. Program. Lang. Syst.*, 4(3):496–517, 1982.

A Code Linearisation

While structured graphs afford a convenient and clear method for constructing graph structures (and reasoning about them!), working with them afterwards can be challenging. In particular, implementing complex transformations in terms of *gfold* is not straightforward. A more pragmatic approach is to take the output of $comp_{\mathsf{G}}$ of type *Graph Code* and transform it into another graph representation, e.g. the graph representation of Hoopl [18], which then allows us to perform data-flow analysis and rewriting.

To illustrate, how to further process the output of our code generator function $comp_{\mathsf{G}}$, we show how to transform a code graph into a linear form with explicit labels and jumps. To this end, we us the following representation of linearised code:

$$
\begin{aligned}
&\textbf{type}\ \mathit{Label} &&= \mathit{Int}\\
&\textbf{data}\ \mathit{Inst} &&= \mathit{PUSH}_{\mathsf{L}}\ \mathit{Int} \mid \mathit{ADD}_{\mathsf{L}} \mid \mathit{THROW}_{\mathsf{L}} \mid \mathit{MARK}_{\mathsf{L}}\ \mathit{Label}\\
& &&\mid\ \mathit{UNMARK}_{\mathsf{L}} \mid \mathit{JUMP}\ \mathit{Label} \mid \mathit{LABEL}\ \mathit{Label}\\
&\textbf{type}\ \mathit{Code}_{\mathsf{L}} &&= [\mathit{Inst}]
\end{aligned}
$$

For each constructor of *Code*, we have a corresponding constructor in the type of instructions *Inst*. Additionally, we also have *JUMP*, representing a jump instruction, and *LABEL*, representing a jump target. Note that in order to have linear code, we have to get rid of the branching of the *MARK* constructor. That is why, we replaced the handler code argument of *MARK* with a label argument.

For the transformation from *Graph Code* to $\mathit{Code}_{\mathsf{L}}$, we need a means to generate fresh labels. To this end, we assume a monad *Fresh* with the following interface to obtain fresh labels and to escape from the monad:

$$
\begin{aligned}
&\mathit{fresh} :: \mathit{Fresh}\ \mathit{Label}\\
&\mathit{runFresh} :: \mathit{Fresh}\ a \to a
\end{aligned}
$$

The linearisation is defined as a general fold over the graph structure:

$$\begin{array}{l} \mathit{linearCode} :: \mathit{Graph}\ \mathit{Code} \rightarrow \mathit{Code}_{\mathsf{L}} \\ \mathit{linearCode}\ c = \mathit{runFresh}\ (\mathit{gfold}\ \mathit{lVar}\ \mathit{lLet}\ \mathit{lAlg}\ c\ [\,]) \end{array}$$

The simplest way to define the transformation is to take $\mathit{Fresh}\ \mathit{Code}_{\mathsf{L}}$ as the result type of the fold. However, since we want to construct a list, we rather want to use an accumulation parameter as well. Hence, the result type is $\mathit{Code}_{\mathsf{L}} \rightarrow \mathit{Fresh}\ \mathit{Code}_{\mathsf{L}}$. The additional argument $[\,]$ to the fold above is the initial value of the accumulator.

Before we look at the components of the fold, we introduce a simple auxiliary operator, which is used to construct lists in a monad:

$$\begin{array}{l} (\langle : \rangle) :: \mathit{Monad}\ m \Rightarrow a \rightarrow m\ [a] \rightarrow m\ [a] \\ \mathit{ins}\ \langle : \rangle\ \mathit{mc} = \mathit{mc} \gg\!= (\lambda c \rightarrow \mathit{return}\ (\mathit{ins} : c)) \end{array}$$

The algebra lAlg has the carrier type $\mathit{Code}_{\mathsf{L}} \rightarrow \mathit{Fresh}\ \mathit{Code}_{\mathsf{L}}$:

$$\begin{array}{ll} \mathit{lAlg} :: \mathit{Code}\ (\mathit{Code}_{\mathsf{L}} \rightarrow \mathit{Fresh}\ \mathit{Code}_{\mathsf{L}}) \rightarrow \mathit{Code}_{\mathsf{L}} \rightarrow \mathit{Fresh}\ \mathit{Code}_{\mathsf{L}} & \\ \mathit{lAlg}\ (\mathit{ADD}\ c) & d = \mathit{ADD}_{\mathsf{L}}\ \langle : \rangle\ c\ d \end{array}$$

$$
\begin{array}{ll}
\mathit{lAlg}\ (\mathit{PUSH}\ n\ c) & d = \mathit{PUSH}_{\mathsf{L}}\ n \mathbin{\langle:\rangle} c\ d \\
\mathit{lAlg}\ \mathit{THROW} & d = \mathit{return}\ [\mathit{THROW}_{\mathsf{L}}] \\
\mathit{lAlg}\ (\mathit{MARK}\ h\ c) & d = \mathit{fresh} \gg\!= \lambda l \to \mathit{MARK}_{\mathsf{L}}\ l \mathbin{\langle:\rangle} (c =\!\ll \mathit{LABEL}\ l \mathbin{\langle:\rangle} h\ d) \\
\mathit{lAlg}\ (\mathit{UNMARK}\ c) & d = \mathit{UNMARK}_{\mathsf{L}} \mathbin{\langle:\rangle} c\ d \\
\mathit{lAlg}\ \mathit{HALT} & d = \mathit{return}\ []
\end{array}
$$

Note that we use the operator $=\!\ll$, which is simply the monadic bind operator $\gg\!=$ with its arguments flipped. The case for *MARK* may need some explanation: We replace the exception handler argument of *MARK* with a fresh label l and continue with the code c. However, we change the accumulator d by putting the instruction $\mathit{LABEL}\ l$ followed by the exception handler code h in front of it.

The components *lVar* and *lLet* deal with the sharing of the graph. For their implementations, we instantiate the type of metavariables in graphs with the type *Label* and turn every metavariable into a jump $\mathit{JUMP}\ l$. However, we make use of the accumulation argument in order to check whether the next instruction is in fact a jump target with the same label l. If so, we can omit the jump:

$$
\begin{array}{l}
\mathit{lVar} :: \mathit{Label} \to \mathit{Code}_{\mathsf{L}} \to \mathit{Fresh}\ \mathit{Code}_{\mathsf{L}} \\
\mathit{lVar}\ l\ (\mathit{LABEL}\ l' : d) \mid l \equiv l' = \mathit{return}\ (\mathit{LABEL}\ l' : d) \\
\mathit{lVar}\ l\ d \qquad\qquad\qquad\quad\ \ = \mathit{return}\ (\mathit{JUMP}\ l : d)
\end{array}
$$

The concrete label l is provided by the *lLet* component of the fold, which creates a fresh label l and passes it to the scope of the let binding. A corresponding jump target $\mathit{LABEL}\ l$ is created just before the code bound by the let binding:

$$
\begin{aligned}
&\mathit{lLet} :: (\mathit{Code}_{\mathsf{L}} \to \mathit{Fresh}\ \mathit{Code}_{\mathsf{L}}) \to (\mathit{Label} \to \mathit{Code}_{\mathsf{L}} \to \mathit{Fresh}\ \mathit{Code}_{\mathsf{L}}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \to \mathit{Code}_{\mathsf{L}} \to \mathit{Fresh}\ \mathit{Code}_{\mathsf{L}} \\
&\mathit{lLet}\ b\ s\ d = \mathit{fresh} \gg\!= \lambda l \to s\ l =\!\ll \mathit{LABEL}\ l\ \langle:\rangle\ b\ d
\end{aligned}
$$

Composing the linearisation with the compiler $\mathit{comp}_{\mathsf{G}}$ then yields a compiler to linearised code:

$$
\begin{aligned}
&\mathit{comp}_{\mathsf{L}} :: \mathit{Expr} \to \mathit{Code}_{\mathsf{L}} \\
&\mathit{comp}_{\mathsf{L}} = \mathit{linearCode} \circ \mathit{comp}_{\mathsf{G}}
\end{aligned}
$$

For example, given the expression $\mathit{Add}\ (\mathit{Catch}\ (\mathit{Val}\ 1)\ (\mathit{Val}\ 2))\ (\mathit{Val}\ 3)$, $\mathit{comp}_{\mathsf{L}}$ produces the following code:

$$
\begin{aligned}
&[\mathit{MARK}_{\mathsf{L}}\ 1, \mathit{PUSH}_{\mathsf{L}}\ 1, \mathit{UNMARK}_{\mathsf{L}}, \mathit{JUMP}\ 0, \mathit{LABEL}\ 1, \mathit{PUSH}_{\mathsf{L}}\ 2, \\
&\ \mathit{LABEL}\ 0, \mathit{PUSH}_{\mathsf{L}}\ 3, \mathit{ADD}_{\mathsf{L}}]
\end{aligned}
$$

For comparison, the code graph produced by $\mathit{comp}_{\mathsf{G}}$ is

$$
\begin{aligned}
&\mathit{Let}\ (\mathit{PUSH}_{\mathsf{G}}\ 3 \triangleright \mathit{ADD}_{\mathsf{G}} \triangleright \mathit{HALT}_{\mathsf{G}})\ (\lambda v \to \mathit{MARK}_{\mathsf{G}}\ (\mathit{PUSH}_{\mathsf{G}}\ 2 \triangleright \mathit{Var}\ v) \triangleright \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{PUSH}_{\mathsf{G}}\ 1 \triangleright \mathit{UNMARK}_{\mathsf{G}} \triangleright \mathit{Var}\ v)
\end{aligned}
$$

Note that if we omitted the first clause of the definition of lVar, then the result would have an additional instruction $\mathit{JUMP}\ 0$ just before $\mathit{LABEL}\ 0$.

B A Monadic Approach

The compiler $comp^{\mathsf{A}}_{\mathsf{T}}$ in Section 3.1 follows a fairly regular recursion scheme. It is *fold* with a function type as result type. Instead of viewing this as such a fold, it can also be seen as a fold with an additional accumulation parameter, viz. the code continuation. Recursion schemes of this form are well studied in automata theory under the name *macro tree transducers* [7]. We will not go into the details of these automata. An important property of macro tree transducers is that they can be transformed (entirely mechanically) into recursive function definitions without accumulation parameters [13]. If there is only a single recursive function, as in our case, we even get a simple fold.

The idea, originally developed by Matsuda et al. [13], is to replace a function f with an accumulation parameter by a function f' that produces a *context* with the property that

$$f\ x\ a = (f'\ x)[a]$$

That is, we obtain the result of the original function f by simply plugging in the accumulation argument in to the context that f' produces.

We shall use free monads in order to represent these contexts. To this end, we modify the type constructor Tree to obtain the type constructor $\mathit{Tree}_{\mathsf{M}}$ of free monads:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Tree}_{\mathsf{M}}\ f\ a = \mathit{Return}\ a \mid \mathit{In}_{\mathsf{M}}\ (f\ (\mathit{Tree}_{\mathsf{M}}\ f\ a)) \\
\textbf{instance}\ \mathit{Functor}\ f \Rightarrow \mathit{Monad}\ (\mathit{Tree}_{\mathsf{M}}\ f)\ \textbf{where} \\
\quad \mathit{return} \qquad\qquad = \mathit{Return} \\
\quad \mathit{Return}\ x \gg\!= f = f\ x \\
\quad \mathit{In}_{\mathsf{M}}\ t \gg\!= f = \mathit{In}_{\mathsf{M}}\ (\mathit{fmap}\ (\lambda s \to s \gg\!= f)\ t)
\end{array}
$$

We start by reformulating the definition of $\mathit{comp}^{\mathsf{A}}_{\mathsf{T}}$ to work with the free monad type instead. To this end, we use an empty type Empty in order to represent the type $\mathit{Tree}\ \mathit{Code}$ above as $\mathit{Tree}_{\mathsf{M}}\ \mathit{Code}\ \mathit{Empty}$:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{comp}^{\mathsf{A}}_{\mathsf{M}} :: \mathit{Expr} \to \mathit{Tree}_{\mathsf{M}}\ \mathit{Code}\ \mathit{Empty} \to \mathit{Tree}_{\mathsf{M}}\ \mathit{Code}\ \mathit{Empty}} \\
\mathit{comp}^{\mathsf{A}}_{\mathsf{M}}\ (\mathit{Val}\ n) & c = & \mathit{PUSH}_{\mathsf{M}}\ n\ c \\
\mathit{comp}^{\mathsf{A}}_{\mathsf{M}}\ (\mathit{Add}\ x\ y) & c = & \mathit{comp}^{\mathsf{A}}_{\mathsf{M}}\ x\ (\mathit{comp}^{\mathsf{A}}_{\mathsf{M}}\ y\ (\mathit{ADD}_{\mathsf{M}}\ c)) \\
\mathit{comp}^{\mathsf{A}}_{\mathsf{M}}\ \mathit{Throw} & c = & \mathit{THROW}_{\mathsf{M}} \\
\mathit{comp}^{\mathsf{A}}_{\mathsf{M}}\ (\mathit{Catch}\ x\ h) & c = & \mathit{MARK}_{\mathsf{M}}\ (\mathit{comp}^{\mathsf{A}}_{\mathsf{M}}\ h\ c)\ (\mathit{comp}^{\mathsf{A}}_{\mathsf{M}}\ x\ (\mathit{UNMARK}_{\mathsf{M}}\ c))
\end{array}
$$

Note that the definition uses smart constructors for the $\mathit{Tree}_{\mathsf{M}}$ type indicated by index M.

The transformation of $comp^{\mathsf{A}}_{\mathsf{M}}$ into a context producing function is straightforward. Since, the function $comp^{\mathsf{A}}_{\mathsf{M}}$ only has one accumulation parameter, the context that we produce only has one type of hole. Therefore, we use the unit type $()$ as the type of holes in the free monad, i.e. $\mathit{Tree}_{\mathsf{M}}\ \mathit{Code}\ ()$ is the type of contexts. Thus the holes in this type of contexts is denoted by $\mathit{return}\ ()$ and we therefore define

$$\mathit{hole} = \mathit{return}\ ()$$

Moreover, plugging an accumulation argument into a context of type $\mathit{Tree}\ \mathit{Code}\ ()$ is achieved using the free monad's bind operator $\gg$. Thus the function $comp^{\mathsf{C}}_{\mathsf{M}}$ that we want to derive from $comp^{\mathsf{A}}_{\mathsf{M}}$ must satisfy the equation

$$comp^{\mathsf{A}}_{\mathsf{M}}\ e\ c = comp^{\mathsf{C}}_{\mathsf{M}}\ e \gg c \qquad \text{for all } e \text{ and } c. \tag{2}$$

We then obtain the definition of $comp^{\mathsf{C}}_{\mathsf{M}}$ from the definition of $comp^{\mathsf{A}}_{\mathsf{M}}$ by replacing all occurrences of the accumulation variable c on the right-hand side with hole and each occurrence of $comp^{\mathsf{A}}_{\mathsf{M}}\ e\ x$ with $comp^{\mathsf{C}}_{\mathsf{M}}\ e \gg x$:

$$comp^{\mathsf{C}}_{\mathsf{M}} :: \mathit{Expr} \to \mathit{Tree}_{\mathsf{M}}\ \mathit{Code}\ ()$$

$$\begin{aligned}
comp^{\mathsf{C}}_{\mathsf{M}}\ (Val\ n) &= PUSH_{\mathsf{M}}\ n\ hole\\
comp^{\mathsf{C}}_{\mathsf{M}}\ (Add\ x\ y) &= comp^{\mathsf{C}}_{\mathsf{M}}\ x \gg comp^{\mathsf{C}}_{\mathsf{M}}\ y \gg ADD_{\mathsf{M}}\ hole\\
comp^{\mathsf{C}}_{\mathsf{M}}\ (Throw) &= THROW_{\mathsf{M}}\\
comp^{\mathsf{C}}_{\mathsf{M}}\ (Catch\ x\ h) &= MARK_{\mathsf{M}}\ (comp^{\mathsf{C}}_{\mathsf{M}}\ h)\ (comp^{\mathsf{C}}_{\mathsf{M}}\ x \gg UNMARK_{\mathsf{M}}\ hole)
\end{aligned}$$

Note that if we follow the transformation rules mechanically the right-hand side for *Catch* should be as follows:

$$MARK_{\mathsf{M}}\ (comp^{\mathsf{C}}_{\mathsf{M}}\ h \gg hole)\ (comp^{\mathsf{C}}_{\mathsf{M}}\ x \gg UNMARK_{\mathsf{M}}\ hole)$$

However, according to the monad laws $c \gg hole = c$ for all c, and thus $comp^{\mathsf{C}}_{\mathsf{M}}\ h \gg hole$ can be replaced by $comp^{\mathsf{C}}_{\mathsf{M}}\ h$.

We then get the final compiler by plugging the *HALT* instruction into the context produced by $comp^{\mathsf{C}}_{\mathsf{M}}$:

$$\begin{aligned}
&comp_{\mathsf{M}} :: Expr \to Tree_{\mathsf{M}}\ Code\ Empty\\
&comp_{\mathsf{M}}\ e = comp^{\mathsf{C}}_{\mathsf{M}}\ e \gg HALT_{\mathsf{M}}
\end{aligned}$$

The virtual machine $exec_{\mathsf{T}}$ can be easily translated into the free monad setting using the following fold operation on $Tree_{\mathsf{M}}$

$$fold_{\mathsf{M}} :: Functor\ f \Rightarrow (f\ r \to r) \to Tree_{\mathsf{M}}\ f\ Empty \to r$$

$$fold_{\mathsf{M}}\ alg\ (In_{\mathsf{M}}\ t) = alg\ (fmap\ (fold_{\mathsf{M}}\ alg)\ t)$$

We can reuse the algebra used in the definition of $exec_{\mathsf{T}}$:

$$\begin{aligned}
&exec_{\mathsf{M}} :: Tree_{\mathsf{M}}\ Code\ Empty \to Stack \to Stack \\
&exec_{\mathsf{M}} = fold_{\mathsf{M}}\ execAlg
\end{aligned}$$

From Equation (2) and the correctness result in Corollary 2 the corresponding result for $comp_{\mathsf{M}}$ is evident:

Corollary 2.

$$exec_{\mathsf{M}}\ (comp_{\mathsf{M}}\ e)\ [\,] = conv\ (eval\ e)$$

So what does this transformation of the compiler into the form of $comp_{\mathsf{M}}$ buy us? It will simplify the reasoning of the graph based compiler by replacing function composition with the monadic bind. In order to make use of this observation, we have to implement the graph based compiler in a monadic style as well. To this end, we turn the type *Graph* into a monad similarly to $Tree_{\mathsf{M}}$:

$$\begin{aligned}
\mathbf{data}\ Graph'_{\mathsf{M}}\ f\ b\ a &= GReturn\ a \\
&\mid GIn_{\mathsf{M}}\ (f\ (Graph'_{\mathsf{M}}\ f\ b\ a))
\end{aligned}$$

$$
\begin{aligned}
&\quad\mid\ \mathit{Let}_{\mathsf{M}}\ (\mathit{Graph}'_{\mathsf{M}}\ f\ b\ a)\ (b \to \mathit{Graph}'_{\mathsf{M}}\ f\ b\ a)\\
&\quad\mid\ \mathit{Var}_{\mathsf{M}}\ b
\end{aligned}
$$

$$\mathbf{newtype}\ \mathit{Graph}_{\mathsf{M}}\ f\ a = \mathit{Graph}_{\mathsf{M}}\ \{\mathit{unGraphM} :: \forall\ b\ .\ \mathit{Graph}'_{\mathsf{M}}\ f\ b\ a\}$$

One can show that, given a strictly positive functor f and any type b $\mathit{Graph}'_{\mathsf{M}}\ f\ b$ forms a monad with the following definitions:

$$
\begin{aligned}
&\mathbf{instance}\ \mathit{Functor}\ f \Rightarrow \mathit{Monad}\ (\mathit{Graph}'_{\mathsf{M}}\ f\ b)\ \mathbf{where}\\
&\quad \mathit{return}\ x = (\mathit{GReturn}\ x)\\
&\quad \mathit{Var}_{\mathsf{M}}\ x \ggg s = \mathit{Var}_{\mathsf{M}}\ x\\
&\quad \mathit{Let}_{\mathsf{M}}\ e\ f \ggg s = \mathit{Let}_{\mathsf{M}}\ (e \ggg s)\ (\lambda x \to f\ x \ggg s)\\
&\quad \mathit{GReturn}\ x \ggg s = s\ x\\
&\quad \mathit{GIn}_{\mathsf{M}}\ t \ggg s = \mathit{GIn}_{\mathsf{M}}\ (\mathit{fmap}\ (\ggg s)\ t)
\end{aligned}
$$

From this one can derive that, given any strictly positive functor f, $\mathit{Graph}_{\mathsf{M}}\ f$ forms a monad as well:

$$
\begin{aligned}
&\mathbf{instance}\ \mathit{Functor}\ f \Rightarrow \mathit{Monad}\ (\mathit{Graph}_{\mathsf{M}}\ f)\ \mathbf{where}\\
&\quad \mathit{return}\ x = \mathit{Graph}_{\mathsf{M}}\ (\mathit{return}\ x)\\
&\quad \mathit{Graph}_{\mathsf{M}}\ g \ggg f = \mathit{Graph}_{\mathsf{M}}\ (g \ggg \mathit{unGraphM} \circ f)
\end{aligned}
$$

We then derive the function $\mathit{comp}^{\mathsf{C}}_{\mathsf{GM}}$ from $\mathit{comp}^{\mathsf{C}}_{\mathsf{M}}$ as in the same way we derived the non-monadic graph-based compiler in Section 3:

$$
\begin{aligned}
&\mathit{comp}^{\mathsf{C}}_{\mathsf{GM}} :: \mathit{Expr} \to \mathit{Graph}'_{\mathsf{M}}\ \mathit{Code}\ b\ ()\\
&\mathit{comp}^{\mathsf{C}}_{\mathsf{GM}}\ (\mathit{Val}\ n) \quad = \quad \mathit{PUSH}_{\mathsf{GM}}\ n\ \mathit{hole}\\
&\mathit{comp}^{\mathsf{C}}_{\mathsf{GM}}\ (\mathit{Add}\ x\ y) = \quad \mathit{comp}^{\mathsf{C}}_{\mathsf{GM}}\ x \gg \mathit{comp}^{\mathsf{C}}_{\mathsf{GM}}\ y \gg \mathit{ADD}_{\mathsf{GM}}\ \mathit{hole}\\
&\mathit{comp}^{\mathsf{C}}_{\mathsf{GM}}\ (\mathit{Throw}) \quad = \quad \mathit{THROW}_{\mathsf{GM}}\\
&\mathit{comp}^{\mathsf{C}}_{\mathsf{GM}}\ (\mathit{Catch}\ x\ h) = \mathit{Let}_{\mathsf{M}}\ \mathit{hole}\ (\lambda e \to \mathit{MARK}_{\mathsf{GM}}\\
&\qquad\qquad\qquad\qquad (\mathit{comp}^{\mathsf{C}}_{\mathsf{GM}}\ h \gg \mathit{Var}_{\mathsf{M}}\ e)\\
&\qquad\qquad\qquad\qquad (\mathit{comp}^{\mathsf{C}}_{\mathsf{GM}}\ x \gg \mathit{UNMARK}_{\mathsf{GM}}\ (\mathit{Var}_{\mathsf{M}}\ e)))
\end{aligned}
$$

And the final compiler is defined as expected:

$$
\begin{aligned}
&\mathit{comp}_{\mathsf{GM}} :: \mathit{Expr} \to \mathit{Graph}_{\mathsf{M}}\ \mathit{Code}\ \mathit{Empty}\\
&\mathit{comp}_{\mathsf{GM}}\ e = \mathit{Graph}_{\mathsf{M}}\ (\mathit{comp}^{\mathsf{C}}_{\mathsf{GM}}\ e \gg \mathit{HALT}_{\mathsf{GM}})
\end{aligned}
$$

In order to define the virtual machine $\mathit{exec}_{\mathsf{G}}$ on $\mathit{Graph}_{\mathsf{M}}$, we define corresponding fold operations:

$$
\mathit{gfold}_{\mathsf{M}} :: \mathit{Functor}\ f \Rightarrow (v \to r) \to (r \to (v \to r) \to r) \to (f\ r \to r)
$$

$$
\begin{aligned}
&\quad \rightarrow \mathit{Graph}_{\mathsf{M}}\ f\ \mathit{Empty} \rightarrow r \\
&\mathit{gfold}_{\mathsf{M}}\ v\ l\ i\ (\mathit{Graph}_{\mathsf{M}}\ g) = \mathit{trans}\ g\ \textbf{where} \\
&\quad \mathit{trans}\ (\mathit{Var}_{\mathsf{M}}\ x) \quad = v\ x \\
&\quad \mathit{trans}\ (\mathit{Let}_{\mathsf{M}}\ e\ f) = l\ (\mathit{trans}\ e)\ (\mathit{trans} \circ f) \\
&\quad \mathit{trans}\ (\mathit{GIn}_{\mathsf{M}}\ t) \quad = i\ (\mathit{fmap}\ \mathit{trans}\ t)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{ufold}_{\mathsf{M}} :: \mathit{Functor}\ f \Rightarrow (f\ r \rightarrow r) \rightarrow \mathit{Graph}_{\mathsf{M}}\ f\ \mathit{Empty} \rightarrow r \\
&\mathit{ufold}_{\mathsf{M}}\ \mathit{alg} = \mathit{gfold}_{\mathsf{M}}\ \mathit{id}\ (\lambda e\ f \rightarrow f\ e)\ \mathit{alg}
\end{aligned}
$$

Again, we reuse the algebra $\mathit{execAlg}$ to define the virtual machine:

$$
\begin{aligned}
&\mathit{exec}_{\mathsf{GM}} :: \mathit{Graph}_{\mathsf{M}}\ \mathit{Code}\ \mathit{Empty} \rightarrow \mathit{Stack} \rightarrow \mathit{Stack} \\
&\mathit{exec}_{\mathsf{GM}} = \mathit{ufold}_{\mathsf{M}}\ \mathit{execAlg}
\end{aligned}
$$

Similar to Theorem 2, we also have a theorem that links $\mathit{ufold}_{\mathsf{M}}$ and $\mathit{fold}_{\mathsf{M}}$ via $\mathit{unravel}_{\mathsf{M}}$ defined as follows:

$$
\begin{aligned}
&\mathit{unravel}_{\mathsf{M}} :: \mathit{Functor}\ f \Rightarrow \mathit{Graph}_{\mathsf{M}}\ f\ a \rightarrow \mathit{Tree}_{\mathsf{M}}\ f\ a \\
&\mathit{unravel}_{\mathsf{M}}\ (\mathit{Graph}_{\mathsf{M}}\ g) = \mathit{unravel}'_{\mathsf{M}}\ g \\
&\mathit{unravel}'_{\mathsf{M}} :: \mathit{Functor}\ f \Rightarrow \mathit{Graph}'_{\mathsf{M}}\ f\ (\mathit{Tree}_{\mathsf{M}}\ f\ a)\ a \rightarrow \mathit{Tree}_{\mathsf{M}}\ f\ a \\
&\mathit{unravel}'_{\mathsf{M}}\ (\mathit{Var}_{\mathsf{M}}\ x) \quad\ = x \\
&\mathit{unravel}'_{\mathsf{M}}\ (\mathit{Let}_{\mathsf{M}}\ e\ f) \quad = \mathit{unravel}'_{\mathsf{M}}\ (f\ (\mathit{unravel}'_{\mathsf{M}}\ e))
\end{aligned}
$$

$$
\begin{aligned}
\mathit{unravel}'_{\mathsf{M}}\;(\mathit{GReturn}\;x) &= \mathit{Return}\;x\\
\mathit{unravel}'_{\mathsf{M}}\;(\mathit{GIn}_{\mathsf{M}}\;t) &= \mathit{In}_{\mathsf{M}}\;(\mathit{fmap}\;\mathit{unravel}'_{\mathsf{M}}\;t)
\end{aligned}
$$

Theorem 4. *Given a strictly positive functor f, some type c, and $alg :: f\; c \to c$, we have*

$$\mathit{ufold}_{\mathsf{M}}\;\mathit{alg} = \mathit{fold}_{\mathsf{M}}\;\mathit{alg} \circ \mathit{unravel}_{\mathsf{M}}$$

This again, yields one half of the correctness proof.

In addition, however, the monadic structure provides us with a generic theorem that facilitates the other half of the correctness proof. The following proposition links the bind operators of the two monads $\mathit{Tree}_{\mathsf{M}}$ and $\mathit{Graph}'_{\mathsf{M}}$:

Proposition 1. *Given $g_1, g_2 :: \forall\, b\,.\, \mathit{Graph}'\; f\; b\; ()$, for any strictly positive functor f, we have*

$$\mathit{unravel}'_{\mathsf{M}}\;(g_1 \gg g_2) = \mathit{unravel}'_{\mathsf{M}}\;g_1 \gg \mathit{unravel}'_{\mathsf{M}}\;g_2$$

Recall that in order to prove the equation

$$\mathit{comp}_{\mathsf{T}} = \mathit{unravel} \circ \mathit{comp}_{\mathsf{G}}$$

we used the equation

$$comp^{\mathsf{A}}_{\mathsf{T}}\ e \circ unravel' = unravel' \circ comp^{\mathsf{A}}_{\mathsf{G}}\ e$$

which we proved by induction. The monadic approach allows us to use a simpler equation in which the unravelling only appears on one side of the equation:

$$comp^{\mathsf{C}}_{\mathsf{M}}\ e = unravel'_{\mathsf{M}}\ (comp^{\mathsf{C}}_{\mathsf{GM}}\ e)$$

For example, the case for $e = Add\ x\ y$ then becomes as follows:

$$
\begin{array}{ll}
 & unravel'_{\mathsf{M}}\ (comp^{\mathsf{C}}_{\mathsf{GM}}\ (Add\ x\ y)) \\
= & \quad \{\ \text{definition of } comp^{\mathsf{C}}_{\mathsf{GM}}\ \} \\
 & unravel'\ (comp^{\mathsf{C}}_{\mathsf{GM}}\ x \gg comp^{\mathsf{C}}_{\mathsf{GM}}\ y \gg ADD_{\mathsf{GM}}\ hole) \\
= & \quad \{\ \text{Proposition 1}\ \} \\
 & unravel'\ (comp^{\mathsf{C}}_{\mathsf{GM}}\ x) \gg unravel'\ (comp^{\mathsf{C}}_{\mathsf{GM}}\ y) \gg unravel'\ (ADD_{\mathsf{GM}}\ hole) \\
= & \quad \{\ \text{definition of } unravel'_{\mathsf{M}}\ \} \\
 & unravel'\ (comp^{\mathsf{C}}_{\mathsf{GM}}\ x) \gg unravel'\ (comp^{\mathsf{C}}_{\mathsf{GM}}\ y) \gg ADD_{\mathsf{M}}\ hole \\
= & \quad \{\ \text{induction hypothesis}\ \} \\
 & comp^{\mathsf{C}}_{\mathsf{M}}\ x \gg comp^{\mathsf{C}}_{\mathsf{M}}\ y \gg ADD_{\mathsf{M}}\ hole \\
= & \quad \{\ \text{definition of } comp^{\mathsf{C}}_{\mathsf{M}}\ \}
\end{array}
$$

$comp_{\mathsf{M}}^{\mathsf{C}}\ (Add\ x\ y)$

22