Pick'n'Fix
Capturing Control Flow in Modular Compilers

Laurence E. Day[1] and Patrick Bahr[2]

[1] Functional Programming Laboratory, University of Nottingham
led@cs.nott.ac.uk
[2] Department of Computer Science, University of Copenhagen
paba@di.ku.dk

Abstract. We present a modular framework for implementing languages with effects and control structures such as loops and conditionals. This framework enables modular definitions of both syntax and semantics as well as modular implementations of compilers and virtual machines. In order to compile control structures, in particular cyclic ones, we employ Oliveira and Cook's purely functional representation of graphs. Moreover, to separate control flow features semantically from other language features, we represent source languages using Johann and Ghani's encoding of generalised algebraic datatypes as fixpoints of higher-order functors. We demonstrate the usage of our modular compiler framework with an extended running example and highlight the extensibility of our modular compiler implementations.

1 Introduction

A compiler is typically decomposed into the various distinct manipulations and translations of the syntax of a source language, such as lexing, parsing and code generation. An additional dimension along which modularity may be exploited is the set of computational features supported by the source language.

In previous work [9, 8], we combined the work of Liang et al. [17] on implementing modular interpreters, and Swierstra's *data types à la carte* [24] to create a modular compiler framework, which we have since built extensively upon. The result to date is a modular framework allowing the definition and implementation of new language features without the modification of any existing definitions. We have demonstrated the implementation of multiple such features, including exceptions, mutable state and functional abstraction.

However, this framework currently falls short of properly capturing the notion of control flow. For example, the compilation module for exception handling currently duplicates a code continuation in order to represent convergent control flow such as seen in conditional statements. Moreover, the framework at present cannot sensibly handle cyclic control structures such as unbounded loops.

The underlying issue responsible for these shortcomings is the fact that the modular target language of the compiler is essentially tree-shaped, and therefore only able to capture tree-shaped control flow.

In this paper, we address this concern by refactoring our modular compilation framework such that it operates on target languages that are *graph* structured instead. This modular graph structure allows us to represent sharing and cycles, solving the issue of code duplication and capturing cyclic control structures respectively.

In particular, the contributions of this paper are as follows:

- We demonstrate the use of Oliveira and Cook's *structured graphs* [21] as the underpinning of the modular compiler target language representation.
- We demonstrate how the additional structure provided by structured graphs allows us to properly compile non-cyclic control structures in a modular fashion without code duplication.
- Most importantly, the graph representation allows us to build compiler modules for circular control structures, which we demonstrate on an imperative language with loops.
- In order to deal with an imperative language, we use a typed representation of the syntax of source languages using Johann and Ghani's fixpoint representation of generalised algebraic datatypes (GADTs) [15].
- We demonstrate our modular compiler framework in action by way of an extended example.

Throughout this paper, we utilise Haskell [18] as both a semantic metalanguage and an implementation language. This paper has been compiled from a literate Haskell source file, and is available in the associated material from the authors' webpages.[3] Due to the space restrictions we elided the modular semantic definitions of the source and target languages presented in this paper. Instead, these semantic definitions can be found in the associated material in the form of a modular interpreter and a modular virtual machine.

2 Compilation à la Carte

Our primary goal is to construct a compiler for imperative languages with control structures such as loops and conditionals. In this section, we illustrate the basic modular compiler framework as presented in previous work [9, 8], before proceeding to identify the problems this approach presents when dealing with control structures.

The fundamental idea is the representation of a data type as the least fixpoint $\mu\ f$ of a functor f (also called the *signature* of the data type):

$$\textbf{data}\ \mu\ f = In\ (f\ \mu\ f)$$

The benefit of defining recursive datatypes in this fashion is that signature functors compose neatly using coproducts:

$$\textbf{data}\ (f \oplus g)\ e = Inl\ (f\ e) \mid Inr\ (g\ e)$$

[3] `http://www.diku.dk/~paba/picknfix.zip`

As a result, we can define fragments of the source language in isolation. Below we define language fragments for arithmetic, comparison operators, sequential composition, state, and conditional control structures:

```
data Arith  a = Val Int                    data State  a = Set Ref a
              | Add a a | Mul a a                        | Get Ref
data Comp a = Equ a a | Lt a a             data If     a = If a a a
data Seq    a = Seq a a                    data While a = While a a
newtype Ref = Ref String   -- reference cells
```

The source language supporting all of the above featues is thus represented by the type $\mu\ (Arith \oplus Comp \oplus Seq \oplus State\ If \oplus While)$.

In order to define functions over datatypes constructed in such a modular manner, we make use of a generic fold operator:

```
fold :: Functor f ⇒ (f c → c) → μ f → c
fold f (In t) = f (fmap (fold f) t)
```

The first argument to the above is called an *f-algebra*, and is intuitively viewed as a directive for recursively processing data constructors. The type c of an f-algebra is called its *carrier* and forms the result type of a fold.

To circumvent the need to manually tag values with *Inl* and *Inr* constructors when introducing elements of a compound functor constructed using $\oplus$, the data type à la carte technique makes use of a typeclass $\triangleleft$, which provides an automatic injection mechanism:

```
class g ◁ f where                  inject :: (g ◁ f) ⇒ g μ f → μ f
  inj :: g a → f a                 inject = In ∘ inj
```

We do not define the instance declarations for $\triangleleft$ here, but instead refer the interested reader to the original work [24]. Suffice it to say that we have $g \triangleleft f$ if and only if f is a sum with g a summand in f.

To further simplify the construction of elements of a fixpoint $\mu\ f$, we make use of *smart constructors* of the following form:

```
If_μ :: (If ◁ f) ⇒ μ f → μ f → μ f → μ f
If_μ c i t = inject (If c i t)
```

For example, we can write the following program in our extensible language:

```
let x = Ref "x"
in While_μ (Get_μ x `Lt_μ` Val_μ 10) (Set_μ x (Get_μ x `Add_μ` Val_μ 1))
```

The program increments the reference cell x until it reaches the value 10.

Next, we show how to use this framework in order to implement a compiler in a modular manner. To this end, we first define the appropriate signatures for the *target* language. For now we only consider the arithmetic and conditional language signatures:

$$\textbf{data}\ ARITH\ e = PUSH\ Int\ e \mid ADD\ e \mid MUL\ e$$
$$\textbf{data}\ COND\ \ e = JPC\ e\ e$$
$$\textbf{data}\ HALT\ \ e = HALT$$

Intuitively, a *PUSH* instruction places its associated integer on top of the current stack, and the *ADD* and *MUL* replace the two topmost elements of the stack by their sum respectively their product. The *JPC* instruction of the conditional signature pops the first integer off the stack and executes its first argument if this integer is non-zero; otherwise it executes the second argument. Finally, the *HALT* instruction simply ends the computation.

Secondly, we identify the structure of the compilation algebra. In this paper, we define our compilers using *continuation-passing style* (CPS), requiring the presence of an accumulator argument:

$$\textbf{class}\ (Functor\ f) \Rightarrow Alg_{Co}\ f\ g\ \textbf{where}$$
$$\quad alg_{Co} :: f\ (\mu\ g \to \mu\ g) \to \mu\ g \to \mu\ g$$

The final compiler is obtained by folding over the compilation algebra defined above and providing the *HALT* instruction as an initial continuation.

$$comp :: (HALT \oslash g, Alg_{Co}\ f\ g) \Rightarrow \mu\ f \to \mu\ g$$
$$comp\ p = fold\ alg_{Co}\ p\ HALT_{\mu}$$

In order to simplify the presentation of our CPS compiler, we define an explicit function application operator (similar to Haskell's standard \$ operator):

$$\textbf{infixr}\ 0 \triangleright$$
$$(\triangleright) :: (a \to b) \to a \to b$$
$$f \triangleright x = f\ x$$

Finally, we have to declare instances of the compilation algebra. The following declaration derives instances for compound source language signatures:

$$\textbf{instance}\ (Alg_{Co}\ f\ h, Alg_{Co}\ g\ h) \Rightarrow Alg_{Co}\ (f \oplus g)\ h\ \textbf{where}$$
$$\quad alg_{Co}\ (Inl\ \ x) = alg_{Co}\ x$$
$$\quad alg_{Co}\ (Inr\ y) = alg_{Co}\ y$$

Then we only need to declare instances for each of the individual source language signatures. We start with arithmetic and conditionals:

$$\textbf{instance}\ (ARITH \oslash g) \Rightarrow Alg_{Co}\ Arith\ g\ \textbf{where}$$
$$\quad alg_{Co}\ (Val\ n)\quad\ \ c = PUSH_{\mu}\ n \triangleright c$$
$$\quad alg_{Co}\ (Add\ x\ y)\ c = x \triangleright y \triangleright ADD_{\mu} \triangleright c$$
$$\quad alg_{Co}\ (Mul\ x\ y)\ c = x \triangleright y \triangleright MUL_{\mu} \triangleright c$$
$$\textbf{instance}\ (COND \oslash g) \Rightarrow Alg_{Co}\ If\ g\ \textbf{where}$$
$$\quad alg_{Co}\ (If\ \ b\ x\ y)\ c = b \triangleright JPC_{\mu}\ (x \triangleright c)\ (y \triangleright c)$$

We can thus instantiate the modular compiler for the implemented language features, e.g.:

$$compArithIf :: \mu\ (Arith \oplus If) \to \mu\ (ARITH \oplus COND \oplus HALT)$$
$$compArithIf = comp$$

There is a subtle issue in the instantiation for *If*, however: the target code produced for *If* first evaluates the Boolean condition before executing the corresponding branch. However, after that, control flow returns to the continuation c. Due to the limitations of the target language, this continuation must be duplicated. A similar duplication occurs in the compilation of exception handlers as seen in previous work [9, 8]

A more pressing issue becomes apparent if we wish to define an instance of the compilation algebra for *While*:

$$\textbf{instance}\ (COND \circledcirc g) \Rightarrow Alg_{Co}\ While\ g\ \textbf{where}$$
$$alg_{Co}\ (While\ b\ lb)\ c = b \triangleright JPC_{\mu}\ (lb \triangleright \dots)\ c$$

What we require in place of the ellipsis is an instruction which jumps back to the check of the condition b, after executing the loop body lb. The tree structure of the target language does not allow us to represent this cyclic control flow.

The core idea we present here in order to fix the issues described above is simple: endow the target language with a graph structure. In the next section, we illustrate how to do this whilst retaining the modular properties of the target language and the compiler implementation.

3 A Graph-Structured Target Language

Typically, a graph-structured target language for a compiler uses explicit jumps and labels (as seen in, for example, Hoopl [22]). This style of representing graphs, whilst close to physical machine implementations, is quite tedious to use and error prone when humans become involved. Instead, we shall use the purely functional representation of graphs by Oliveira and Cook [21], dubbed *structured graphs*. In short, structured graphs provide a representation of term graphs, which uses an elegant encoding of sharing and cyclicity using *parametric higher-order abstract syntax* [7]. This representation provides a simple interface to construct graphs in a compositional fashion (at the cost of a more complicated and restrictive interface for 'consuming' them).

3.1 From Fixpoints to Graphs

Structured graphs, at their core, represent *term graphs* using mutually recursive let bindings. To this end the definition of structured graphs extends the definition of fixpoints by two additional constructors, *Var* and *Mu*, for metavariables and mutually recursive bindings, respectively:

$$\begin{aligned}\textbf{data}\ Graph_{\mathsf{T}}\ f\ v = {} & Var\ v \\ \mid {} & Mu\ ([v] \to [Graph_{\mathsf{T}}\ f\ v]) \\ \mid {} & In_{\mathsf{G}}\ (f\ (Graph_{\mathsf{T}}\ f\ v))\end{aligned}$$

The additional type variable v is used as the type for the metavariables in the graph. *Mu* represents binders using higher-order abstract syntax (HOAS). In order to allow mutually recursive bindings, *Mu* is a function that takes a list of metavariables and returns a list of corresponding graphs. The intention is that the i-th metavariable in the argument list is bound to the i-th graph in the result list. The entry point is the first graph in the result list.

In this paper, we only use the above *Mu* constructor in two distinct cases, viz. non-recursive let bindings and fixpoints over a single argument, which we express as follows:

$$\begin{aligned}
&\mathit{letx} :: \mathit{Graph}_{\mathsf{T}}\ f\ v \to (v \to \mathit{Graph}_{\mathsf{T}}\ f\ v) \to \mathit{Graph}_{\mathsf{T}}\ f\ v\\
&\mathit{letx}\ g\ f = \mathit{Mu}\ (\lambda{\sim}(_ : x : _) \to [f\ x, g])\\
&\mathit{mu} :: (v \to \mathit{Graph}_{\mathsf{T}}\ f\ v) \to \mathit{Graph}_{\mathsf{T}}\ f\ v\\
&\mathit{mu}\ f = \mathit{Mu}\ (\lambda{\sim}(x : _) \to [f\ x])
\end{aligned}$$

With these two combinators we may represent a non-recursive let binding "**let** $x = b$ **in** s" as $\mathit{letx}\ b\ (\lambda x \to s)$, and a fixpoint of f as $\mathit{mu}\ f$.

Structured graphs use a restricted form of HOAS called *parametric HOAS* [7]. When constructing structured graphs the type v of metavariables is left polymorphic. To ensure this, structured graphs are wrapped in the following type, which enforces the parametric polymorphism of the metavariables type v:

$$\textbf{newtype}\ \mathit{Graph}\ f = \mathit{MkGraph}\ (\forall\ v\ .\ \mathit{Graph}_{\mathsf{T}}\ f\ v)$$

The parametricity of the type of metavariables ensures that the function argument passed to the *Mu* constructor is indeed only used for defining binders.

In analogy to smart constructors for the fixpoint type constructor $\mu\ \cdot$, we use smart constructors to simplify the construction of graphs. To this end we transcribe *inject* from fixpoints to graphs:

$$\begin{aligned}
&\mathit{inject}_{\mathsf{G}} :: (f \mathbin{\triangleleft\!\circledcirc} g) \Rightarrow f\ (\mathit{Graph}_{\mathsf{T}}\ g\ a) \to \mathit{Graph}_{\mathsf{T}}\ g\ a\\
&\mathit{inject}_{\mathsf{G}} = \mathit{In}_{\mathsf{G}} \circ \mathit{inj}
\end{aligned}$$

Using $\mathit{inject}_{\mathsf{G}}$, we obtain smart constructors such as the following:

$$\begin{aligned}
&\mathit{PUSH}_{\mathsf{G}} :: (\mathit{ARITH} \mathbin{\triangleleft\!\circledcirc} f) \Rightarrow \mathit{Int} \to \mathit{Graph}_{\mathsf{T}}\ f\ v \to \mathit{Graph}_{\mathsf{T}}\ f\ v\\
&\mathit{PUSH}_{\mathsf{G}}\ n\ c = \mathit{inject}_{\mathsf{G}}\ (\mathit{PUSH}\ n\ c)
\end{aligned}$$

3.2 Compiling To Graphs

We can now refactor the carrier of the compilation algebra to reflect the usage of structured graphs instead of least fixpoints:

$$\begin{aligned}
&\textbf{class}\ \mathit{Functor}\ f \Rightarrow \mathit{Alg}^{\mathsf{G}}_{\mathit{Co}}\ f\ g\ \textbf{where}\\
&\quad \mathit{alg}^{\mathsf{G}}_{\mathit{Co}} :: f\ (\mathit{Graph}_{\mathsf{T}}\ g\ v \to \mathit{Graph}_{\mathsf{T}}\ g\ v) \to \mathit{Graph}_{\mathsf{T}}\ g\ v \to \mathit{Graph}_{\mathsf{T}}\ g\ v
\end{aligned}$$

As before the type class is easily lifted to coproducts:

$$\textbf{instance}\ (Alg^{\mathsf{G}}_{Co}\ f\ g, Alg^{\mathsf{G}}_{Co}\ h\ g) \Rightarrow Alg^{\mathsf{G}}_{Co}\ (f \oplus h)\ g\ \textbf{where}$$
$$\quad alg^{\mathsf{G}}_{Co}\ (Inl\ x) = alg^{\mathsf{G}}_{Co}\ x$$
$$\quad alg^{\mathsf{G}}_{Co}\ (Inr\ y) = alg^{\mathsf{G}}_{Co}\ y$$

Note that we use the $Graph_{\mathsf{T}}$ type constructor instead of $Graph$. As a general rule of thumb, we use $Graph_{\mathsf{T}}$ in order to build up graphs, and after that we use the $MkGraph$ constructor to construct a graph of type $Graph\ g$. The rank 2 polymorphic type of $MkGraph$ ensures that the construction of the underlying graph of type $Graph_{\mathsf{T}}$ was indeed polymorphic in the type v of metavariables:

$$comp_{\mathsf{G}} :: (Alg^{\mathsf{G}}_{Co}\ f\ g, HALT \circledcirc g) \Rightarrow \mu\ f \rightarrow Graph\ g$$
$$comp_{\mathsf{G}}\ e = MkGraph\ (fold\ alg^{\mathsf{G}}_{Co}\ e\ HALT_{\mathsf{G}})$$

The implementation of the compiler for $ARITH$ is analogous to the fixpoint version from section 2. The only aspect that has to be changed is that we use the smart constructors for graphs:

$$\textbf{instance}\ (ARITH \circledcirc g) \Rightarrow Alg^{\mathsf{G}}_{Co}\ Arith\ g\ \textbf{where}$$
$$\quad alg^{\mathsf{G}}_{Co}\ (Val\ n)\quad c = PUSH_{\mathsf{G}}\ n \triangleright c$$
$$\quad alg^{\mathsf{G}}_{Co}\ (Add\ x\ y)\ c = x \triangleright y \triangleright ADD_{\mathsf{G}} \triangleright c$$
$$\quad alg^{\mathsf{G}}_{Co}\ (Mul\ x\ y)\ c = x \triangleright y \triangleright MUL_{\mathsf{G}} \triangleright c$$

In principle, we may transcribe the compiler definition for If in the same simple way. However, we can exploit the sharing capabilities that the graph structure affords us:

$$\textbf{instance}\ (COND \circledcirc g) \Rightarrow Alg^{\mathsf{G}}_{Co}\ If\ g\ \textbf{where}$$
$$\quad alg^{\mathsf{G}}_{Co}\ (If\ b\ p_1\ p_2)\ c = letx\ c$$
$$\qquad (\lambda v \rightarrow b \triangleright JPC_{\mathsf{G}}\ (p_1 \triangleright Var\ v) \triangleright p_2 \triangleright Var\ v)$$

Instead of placing the code continuation c directly into the generated code (and thereby duplicating c), we bind c to the metavariable v, which is then used instead of c, thus avoiding the duplication.

Likewise, for compiling loops, we make use of the graph structure, too. This time, however, we need to construct a cycle, for which we use the mu combinator:

$$\textbf{instance}\ (COND \circledcirc g) \Rightarrow Alg^{\mathsf{G}}_{Co}\ While\ g\ \textbf{where}$$
$$\quad alg^{\mathsf{G}}_{Co}\ (While\ b\ lb)\ c = mu\ (\lambda v \rightarrow b \triangleright JPC_{\mathsf{G}}\ (lb \triangleright Var\ v) \triangleright c)$$

Next, we give the definition for compiling the signatures $State$ and $Comp$. To do this we need corresponding instructions in the target language:

$$\textbf{data}\ STATE\ e = GET\ Ref\ e \mid SET\ Ref\ e$$
$$\textbf{data}\ COMP\ \ e = EQ\ e \mid LT\ e$$

The corresponding compiler definitions are straightforward:

$$\textbf{instance}\ (STATE \mathbin{\triangleleft\!\circledcirc} g) \Rightarrow Alg^{\mathsf{G}}_{Co}\ State\ g\ \textbf{where}$$
$$\quad alg^{\mathsf{G}}_{Co}\ (Get\ v)\quad c = GET_{\mathsf{G}}\ v \triangleright c$$
$$\quad alg^{\mathsf{G}}_{Co}\ (Set\ v\ e)\ c = e \triangleright SET_{\mathsf{G}}\ v \triangleright c$$
$$\textbf{instance}\ (COMP \mathbin{\triangleleft\!\circledcirc} g) \Rightarrow Alg^{\mathsf{G}}_{Co}\ Comp\ g\ \textbf{where}$$
$$\quad alg^{\mathsf{G}}_{Co}\ (Equ\ x\ y)\ c = x \triangleright y \triangleright EQ_{\mathsf{G}} \triangleright c$$
$$\quad alg^{\mathsf{G}}_{Co}\ (Lt\ x\ y)\quad c = x \triangleright y \triangleright LT_{\mathsf{G}} \triangleright c$$

Finally, we give the definition for compiling sequential composition:

$$\textbf{instance}\ Alg^{\mathsf{G}}_{Co}\ Seq\ g\ \textbf{where}$$
$$\quad alg^{\mathsf{G}}_{Co}\ (Seq\ x\ y)\ c = x \triangleright y \triangleright c$$

To see the resulting compiler in action, we consider the following example source program that computes the factorial:

$$\textbf{type}\ Source = Arith \oplus State \oplus Seq \oplus While \oplus Comp$$
$$fac :: \mu\ Source$$
$$fac = Set_\mu\ y\ (Val_\mu\ 1)\ `Seq_\mu`\ While_\mu\ (Val_\mu\ 0\ `Lt_\mu`\ Get_\mu\ x)$$
$$\qquad (Set_\mu\ y\ (Get_\mu\ y\ `Mul_\mu`\ Get_\mu\ x)\ `Seq_\mu`$$
$$\qquad\ Set_\mu\ x\ (Get_\mu\ x\ `Add_\mu`\ Val_\mu\ (-1)))$$
$$\quad \textbf{where}\ x = Ref\ \texttt{"x"}; y = Ref\ \texttt{"y"}$$

In order to determine the target language, we need only sum up the constraints on the target signatures in the declarations of Alg^{G}_{Co} for the individual source language features:

$$\textbf{type}\ Target = ARITH \oplus STATE \oplus COND \oplus COMP \oplus HALT$$
$$compSource :: \mu\ Source \to Graph\ Target$$
$$compSource = comp_{\mathsf{G}}$$

Note that Haskell's type system makes sure that the target language has the instructions required to compile the source language. For instance, if we forget to include *COMP* in the target language, Haskell's type checker complains:

```
No instance for (COMP :<: HALT) arising from a use of ‘comp’
```

Applying *compSource* to *fac* returns the following graph:

```
PUSH 1; SET y; [v1 -> PUSH 0; GET x; LT; JPC (GET y; GET x; MUL;
  SET y; GET x; PUSH (-1); ADD; SET x; v1); HALT]
```

The code inside the brackets corresponds to bindings in the graph structure constructed by *Mu*: the metavariable `v1` is bound to the code to the right of the `->` arrow, which in turn contains a reference to `v1`. Thus, as expected, the output code graph has a single cycle, corresponding to the loop of the source program.

4 Typing the Source Language

We highlight at this point that all instances of source languages we have considered thus far have been defined as a single fixpoint of a signature functor, even though said signature can be compound. We mention this because at present there are no *restrictions* on how to combine these different language fragments, permitting all manner of ill-formed source programs via the abuse of syntax. For example, we can write:

$$While_\mu\ (Set_\mu\ (Ref\ \texttt{"x"})\ (Val_\mu\ 10))\ (Get_\mu\ (Ref\ \texttt{"x"})\ `Add_\mu`\ Val_\mu\ 1)$$

We need to be able to cleanly separate (at least) between *statements*, which can be combined using sequential composition, conditionals and loops, and *expressions*, which can be combined by arithmetic and Boolean operators.

4.1 Splitting the Source Language

In order to split the source language into different syntactic categories – while retaining its modular properties – we use Johann and Ghani's initial algebra semantics of GADTs [15] and combine it with data types à la carte in the style of Bahr and Hvitved [5]. The underlying idea is to annotate each node of the tree type with the syntactic category it resides in. To this end, each signature functor gets an additional type argument (and thus ceases to be a functor). For example, using Haskell's GADT syntax, *Arith* is now defined as follows:

```
data Exp  -- type index for expressions
data Arith e l where Val :: Int                 → Arith e Exp
                     Add :: e Exp → e Exp → Arith e Exp
                     Mul :: e Exp → e Exp → Arith e Exp
```

Note that we define an empty datatype *Exp* as a label – or more precisely, an index – for expressions. The *Arith* signature is simple; each operator only takes expressions and returns expressions. More interesting is the definition of the signature for assigning and dereferencing reference cells:

```
data Stmt  -- type index for statements
data State e l where Get :: Ref           → State e Exp
                     Set :: Ref → e Exp → State e Stmt
```

Note that the *Get* constructor builds an expression, while the *Set* constructor takes an expression and builds a statement.

As we have already noted, the above indexed signatures are no longer Haskell functors: instead of mapping types to types, they map functors to functors (and natural transformations to natural transformations). In the language of Johann and Ghani [15], these signatures are *higher-order* functors.

The type constructors $\oplus$ and $\mu\ \cdot$ are easily adjusted to this setting by also equipping them with an additional type argument:

$$\textbf{data}\ (f \boxplus g)\ (h :: * \to *)\ e = Inl_{\mathsf{H}}\ (f\ h\ e) \mid Inr_{\mathsf{H}}\ (g\ h\ e)$$
$$\textbf{data}\ \mu_{\mathsf{H}}\ f\ i = In_{\mathsf{H}}\ (f\ (\mu_{\mathsf{H}}\ f)\ i)$$

As expected, the fixpoint of a higher-order functor is itself a type function of kind $* \to *$ (in other words, a family of types). In the case of the syntax trees for our target language, this family comprises the different syntactic categories we want to represent. Concretely, $\mu_{\mathsf{H}}\ (Arith \boxplus State)\ Exp$ is the type of expressions over signature $Arith \boxplus State$, whereas $\mu_{\mathsf{H}}\ (Arith \boxplus State)\ Stmt$ is the corresponding type of statements.

As we have just introduced typed syntax trees, we make use of the infrastructure in order to also keep track of the object language typing. Our language is simple enough such that it suffices to add a type for Boolean expressions:

$$\textbf{data}\ BExp$$
$$\begin{aligned}\textbf{data}\ Comp\ e\ l\ \textbf{where}\ &Equ :: e\ Exp \to e\ Exp \to Comp\ e\ BExp\\ &Lt\ \ :: e\ Exp \to e\ Exp \to Comp\ e\ BExp\end{aligned}$$

Signatures for control structures are defined in the same style:

$$\begin{aligned}&\textbf{data}\ If\ e\ l\ \textbf{where}\ \ If :: e\ BExp \to e\ Stmt \to e\ Stmt \to If\ e\ Stmt\\ &\textbf{data}\ While\ e\ l\ \textbf{where}\ While :: e\ BExp \to e\ Stmt \to While\ e\ Stmt\\ &\textbf{data}\ Seq\ e\ l\ \textbf{where}\ Seq :: e\ Stmt \to e\ Stmt \to Seq\ e\ Stmt\end{aligned}$$

We now have the infrastructure to define and combine signatures. In the next section we shall see that the machinery to define folds on fixpoints and to define smart constructors carries over to the setting of higher-order functors easily.

4.2 Folds and Smart Constructors

The representation of modular GADTs that we use here is based on fixpoints of higher-order functors in the style of Johann and Ghani [15]. The key to this representation is that the higher-order functors we make use of here have slightly 'less' structure than what one would typically assume of such constructs: our versions only provide a mapping from natural transformations to natural transformations:

$$\textbf{class}\ HFunctor\ f\ \textbf{where}\ hfmap :: (g \dot{\to} h) \to f\ g \dot{\to} f\ h$$

where natural transformations are defined as follows:

$$\textbf{type}\ f \dot{\to} g = \forall\ i\ .\ f\ i \to g\ i$$

In the language of Johann and Ghani [15], these higher-order functors map functors of kind $| * | \to *$ to functors of kind $| * | \to *$, which are (conveniently enough) exactly what is needed to represent GADTs.

Instance declarations for *HFunctor* are defined in as straightforward a manner as their *Functor* counterparts. For example, to instantiate *State*:

```
instance HFunctor State where
  hfmap f (Get v)   = Get v
  hfmap f (Set v x) = Set v (f x)
```

Using this structure, we can define folds on fixpoints of higher-order functors. Since the signatures are indexed (as are their fixpoints), it follows that the algebras that are used to define these folds are indexed too. More precisely, given a higher-order functor f and a type constructor $c :: * \to *$, an f-algebra with carrier c is a natural transformation of type $f\ c \dot{\to} c$. Apart from the typing, the implementation of higher-order folds is the identical to that of ordinary Haskell functors:

```
fold_H :: HFunctor f ⇒ (f c ·→ c) → μ_H f ·→ c
fold_H f (In_H t) = f (hfmap (fold_H f) t)
```

The definition of the typeclass $\circledlt$ is also lifted to the setting of higher-order functors with minimum hassle:

```
class (sub :: (* → *) → * → *) ◁ sup where
  inj_H :: sub a ·→ sup a
```

The instance declarations for $\circledlt$ are transcribed one-to-one to the typeclass $\boxed{\lhd}$ without surprises, which in turn gives us the corresponding higher-order injection function:

```
inject_H :: (g ◁ f) ⇒ g (μ_H f) ·→ μ_H f
inject_H = In_H ∘ inj_H
```

As before, we assume that each constructor of a higher-order signature functor comes equipped with a corresponding smart constructor, e.g.

```
While_H :: (While ◁ f) ⇒ μ_H f BExp → μ_H f Stmt → μ_H f Stmt
While_H x y = inject_H (While x y)
```

Given these smart constructors, we can write a typed version of the factorial program from section 3 as follows:

```
fac :: μ_H (Arith ⊞ State ⊞ Seq ⊞ While ⊞ Comp) Stmt
fac = Set_H y (Val_H 1) `Seq_H` While_H (Val_H 0 `Lt_H` Get_H x)
        (Set_H y (Get_H y `Mul_H` Get_H x) `Seq_H`
         Set_H x (Get_H x `Add_H` Val_H (−1)))
  where x = Ref "x"; y = Ref "y"
```

4.3 Adapting the Modular Compiler

In this section we briefly describe the changes necessary to adapt the modular compiler implementation from section 3 to this typed setting.

The key concern is in coping with the fact that algebras are now natural transformations rather than functions, and that algebras must now honour the typing that the higher-order functors declare. For example, if we were to write an interpreter, the arguments of Lt must evaluate to integers and Lt itself must evaluate to a Boolean.

For the definition of the compiler the typing is of secondary concern, since the target language is untyped (we consider typing necessary to ensure the well-formedness of *source* programs). In order to discard the typing information when no longer required we introduce a type constructor K, which yields the following definition of the compilation algebra:

$$\begin{array}{l}\textbf{newtype}\ K\ a\ i = K\ \{\mathit{unK} :: a\}\\ \textbf{class}\ (\mathit{HFunctor}\ f) \Rightarrow \mathit{Alg}^{\mathsf{G}}_{\mathit{Co}}\ f\ g\ \textbf{where}\\ \quad \mathit{alg}^{\mathsf{G}}_{\mathit{Co}} :: f\ (K\ (\mathit{Graph}_{\mathsf{T}}\ g\ v \to \mathit{Graph}_{\mathsf{T}}\ g\ v))\\ \qquad\qquad \dot{\to}\ K\ (\mathit{Graph}_{\mathsf{T}}\ g\ v \to \mathit{Graph}_{\mathsf{T}}\ g\ v)\end{array}$$

To simplify instance declarations for the compilation algebra, we remove the occurrence of K in the result type:

$$\begin{array}{l}\textbf{class}\ (\mathit{HFunctor}\ f) \Rightarrow \mathit{Alg}^{\mathsf{G}}_{\mathit{Co}}\ f\ g\ \textbf{where}\\ \quad \mathit{alg}^{\mathsf{G}}_{\mathit{Co}} :: f\ (K\ (\mathit{Graph}_{\mathsf{T}}\ g\ v \to \mathit{Graph}_{\mathsf{T}}\ g\ v))\ i \to \mathit{Graph}_{\mathsf{T}}\ g\ v \to \mathit{Graph}_{\mathsf{T}}\ g\ v\end{array}$$

In order to recover an algebra of the correct type, we compose $\mathit{alg}^{\mathsf{G}}_{\mathit{Co}}$ with K:

$$K \circ \mathit{alg}^{\mathsf{G}}_{\mathit{Co}} :: f\ (K\ (\mathit{Graph}_{\mathsf{T}}\ g\ v \to \mathit{Graph}_{\mathsf{T}}\ g\ v)) \dot{\to} K\ (\mathit{Graph}_{\mathsf{T}}\ g\ v \to \mathit{Graph}_{\mathsf{T}}\ g\ v)$$

The final compiler is thus defined as follows:

$$\begin{array}{l}\mathit{comp}_{\mathsf{G}} :: (\mathit{Alg}^{\mathsf{G}}_{\mathit{Co}}\ f\ g, \mathit{HALT} \triangleleft g) \Rightarrow \mu_{\mathsf{H}}\ f\ i \to \mathit{Graph}\ g\\ \mathit{comp}_{\mathsf{G}}\ e = \mathit{MkGraph}\ (\mathit{unK}\ (\mathit{fold}_{\mathsf{H}}\ (K \circ \mathit{alg}^{\mathsf{G}}_{\mathit{Co}})\ e)\ \mathit{HALT}_{\mathsf{G}})\end{array}$$

Here we use unK to turn the result of the fold – of type $K\ (\mathit{Graph}_{\mathsf{T}}\ g\ v \to \mathit{Graph}_{\mathsf{T}}\ g\ v)\ i$ – into a function of type $\mathit{Graph}_{\mathsf{T}}\ g\ v \to \mathit{Graph}_{\mathsf{T}}\ g\ v$. This function is then applied to a singleton HALT instruction, which serves as the initial continuation. All that remains is to change the instance declarations for the compilation algebra in such a way that they fit this new interface.

The lifting to coproducts now uses the type constructor $\boxplus$ instead of $\oplus$:

$$\begin{array}{l}\textbf{instance}\ (\mathit{Alg}^{\mathsf{G}}_{\mathit{Co}}\ f\ g, \mathit{Alg}^{\mathsf{G}}_{\mathit{Co}}\ h\ g) \Rightarrow \mathit{Alg}^{\mathsf{G}}_{\mathit{Co}}\ (f \boxplus h)\ g\ \textbf{where}\\ \quad \mathit{alg}^{\mathsf{G}}_{\mathit{Co}}\ (\mathit{Inl}_{\mathsf{H}}\ x) = \mathit{alg}^{\mathsf{G}}_{\mathit{Co}}\ x\\ \quad \mathit{alg}^{\mathsf{G}}_{\mathit{Co}}\ (\mathit{Inr}_{\mathsf{H}}\ y) = \mathit{alg}^{\mathsf{G}}_{\mathit{Co}}\ y\end{array}$$

The changes to the instance declarations for the individual language fragments are syntactically very simple: we just need to insert the constructor K for each 'recursive' argument of the language construct. For example, for the *While* language fragment we define the following:

$$\begin{array}{l}\textbf{instance}\ (\mathit{COND} \triangleleft g) \Rightarrow \mathit{Alg}^{\mathsf{G}}_{\mathit{Co}}\ \mathit{While}\ g\ \textbf{where}\\ \quad \mathit{alg}^{\mathsf{G}}_{\mathit{Co}}\ (\mathit{While}\ (K\ b)\ (K\ \mathit{lb}))\ c = \mathit{mu}\ (\lambda v \to b \triangleright \mathit{JPC}_{\mathsf{G}}\ (\mathit{lb} \triangleright \mathit{Var}\ v) \triangleright c)\end{array}$$

4.4 Modularity

The use of higher-order functors may appear to be overkill at first glance, since our goal is simply to separate expressions from statements. Higher-order functors allow us to encode GADTs, but the relation between expressions and statements is much simpler: expressions may occur in statements, but not vice versa. However, we argue – similarly to [5] – that the higher-order functor representation is much better suited to modular definitions.

If we were to retain the use of functors but still wished to discriminate between expressions and statements, we would need to make the expression language a parameter of the *statement* language, e.g.:

$$\begin{aligned}
&\textbf{data } \mathit{If} \quad \mathit{exp}\ \mathit{stmt} = \mathit{If}\ \mathit{exp}\ \mathit{stmt}\ \mathit{stmt}\\
&\textbf{data } \mathit{While}\ \mathit{exp}\ \mathit{stmt} = \mathit{While}\ \mathit{exp}\ \mathit{stmt}\\
&\textbf{type } \mathit{Sig} = \mathit{If}\ (\mu\ \mathit{Arith}) \oplus \mathit{While}\ (\mu\ \mathit{Arith})\\
&\textbf{type } \mathit{Lang} = \mu\ \mathit{Sig}
\end{aligned}$$

This approach complicates the composition of signatures, since we would then need to ensure that each statement signature received the same expression language argument. Moreover, signatures could then no longer be extended compositionally: if we were to extend the signature *Sig* with comparison operators, we could not reuse the definition of *Sig*. We would be forced to define:

$$\textbf{type } \mathit{Sig}' = \mathit{If}\ (\mu\ (\mathit{Arith} \oplus \mathit{Comp})) \oplus \mathit{While}\ (\mu\ (\mathit{Arith} \oplus \mathit{Comp}))$$

Modular algebra definitions become markedly more difficult as well, as we would need to operate over two 'levels' of modularity, namely on the expression and the statement level.

In our opinion, the use of higher-order functors makes modular definitions far simpler and more concise, as we need only deal with one level of modularity. No matter whether we want to extend the statement or the expression language, we simply use the coproduct $\boxplus$ to add the desired signature. Further, as it stands we can exploit the open nature of the kind system, i.e. if we want to add floating point expressions, we simply introduce a new type index *FPExp*:

$$\begin{aligned}
&\textbf{data } \mathit{FPExp} \quad \text{-- type index for floating point expressions}\\
&\textbf{data } \mathit{Div}\ e\ l\ \textbf{where } \mathit{Div} :: e\ \mathit{FPExp} \to e\ \mathit{FPExp} \to \mathit{Div}\ e\ \mathit{FPExp}\\
&\textbf{type } \mathit{Lang} = \mu_{\mathsf{H}}\ (\mathit{Arith} \boxplus \mathit{While} \boxplus \mathit{If} \boxplus \mathit{State} \boxplus \mathit{Div})\ \mathit{Stmt}
\end{aligned}$$

In turn, our modular algebra definitions remain as simple as in the untyped case. For example, assuming that we have a suitable signature *DIV* for the target language, we may extend the compiler for the new floating point operator by the following declaration:

$$\begin{aligned}
&\textbf{instance } (\mathit{DIV} \circledS\, g) \Rightarrow \mathit{Alg}^{\mathsf{G}}_{Co}\ \mathit{Div}\ g\ \textbf{where}\\
&\quad \mathit{alg}^{\mathsf{G}}_{Co}\ (\mathit{Div}\ (K\ x)\ (K\ y))\ c = x \triangleright y \triangleright \mathit{DIV}_{\mathsf{G}} \triangleright c
\end{aligned}$$

5 Discussion

5.1 Future Research Avenues

Additional Computational Constructs We would like to describe other computational features in the manner that we have presented in order to further increase the expressive power of a modular source language. In particular, we are interested in implementing I/O, explicit parallelism and continuations in this manner, and exploring how their semantics interacts with the existing framework.

Data-Flow Analysis and Optimisations A natural extension of our work is to implement data-flow analysis and optimisations in a modular style as well. A good starting point for extending our work in this direction is *Hoopl*, the *Higher Order Optimisation Library* of Ramsey et al. [22]. Hoopl is a Haskell library that allows compiler implementers to define data-flow analyses and implement optimising transformations that are informed by this analysis. Modular implementations of optimising transformations can be achieved using the same techniques as presented in this paper. In particular, data-flow analysis and the underlying lattice structures can be defined in a modular way for (at least) standard textbook analyses.

Testing and Reasoning An important property of a compiler is its trustworthiness. Does it perform only semantics preserving transformations? Establishing such trustworthiness in a modular fashion as well still remains a considerable challenge. However, using the same techniques as presented here, automatic test case generation (i.e. generation of input programs and initial configurations) can be implemented in a modular fashion. Rigorous and machine-checked correctness proofs, however, require new reasoning techniques that work in a modular setting. There is a growing interest in formalising programming language metatheory in a modular fashion [11, 10, 23]. However, building modular proofs of compiler correctness have to deal with additional difficulties. Such proofs have to be modular along both the source and target language as well as computational effects. As the work of Delaware et al. [10] shows, modular reasoning about effects already becomes a considerable obstacle for type soundness proofs.

5.2 Related Work

Modular Compilers William Harrison's doctoral thesis [14] focuses on the topic of modular compilation by defining 'reusable compiler building blocks'. These blocks are presented using two distinct approaches, one of which is *monadic code generators*, which are closely related to our own research. Also contained within each block is a set of equations defining the 'compilation semantics' of the feature in question, and the compiler is verified correct by formulating and proving relations between the standard and compilation semantics.

Marcos Viera's doctoral thesis [25] presents the usage of attribute grammars as the vehicle for defining extensible programming languages. In short, individual fragments of an attribute grammar can describe individual features of a

language, and compositionality is guaranteed via the Haskell type system. There is also a number of standalone attribute grammar systems with particular focus on extensible language implementation such as LISA [19], JastAdd [12] and Silver [28].

Graph Representations for Compilers As previously mentioned, the data-flow analysis library Hoopl [22] uses a graph representation based on explicit labels and jumps. More recently, Bahr [3] demonstrated the use of structured graphs for representing the target language of a compiler. However, Bahr's work is focused on compiler verification and does not consider cyclic graphs. Moreover, neither Ramsey et al. [22] nor Bahr [3] consider the problem of modularising the target language.

Recursion Schemes The recursion schemes used to define the compiler and the semantics heavily influences how compiler and semantics can be extended. More structured recursion schemes derived from tree automata [2, 4] and attribute grammars [27, 26] offer more freedom to *replace* parts of modular definition as opposed to only being able to *extend* them. In the same direction goes the work of Kimmell et al. [16] and Frisby et al. [13] who introduce algebra combinators such as switch and sequence algebras to compose algebras.

Modular Syntax The use of higher-order functors to represent indexed datatypes and families of mutually recursive datatypes stems from Johann and Ghani [15]. Yakushev et al. [29] applied this technique to generic programming. Bahr and Hvitved [5, 6] employed fixpoints of higher-order functors to represent modular syntax and semantics, and combined it with parametric HOAS. Axelsson [1] introduces a different approach to modular well-typed definitions of syntax and semantics: he developed an applicative encoding of the syntax that makes heavy use of type indexing to describe the signature of individual language features. Oliveira and Löh [20] used structured graphs to represent source languages. To this end they combined structured graphs with the typing mechanism of indexed types and modular definitions using type classes.

References

[1] Axelsson, E.: A generic abstract syntax model for embedded languages. In: ICFP (2012)
[2] Bahr, P.: Modular tree automata. In: MPC (2012)
[3] Bahr, P.: Proving correctness of compilers using structured graphs. In: FLOPS. vol. 8475 (2014)
[4] Bahr, P., Day, L.E.: Programming macro tree transducers. In: WGP (2013)
[5] Bahr, P., Hvitved, T.: Compositional data types. In: WGP (2011)
[6] Bahr, P., Hvitved, T.: Parametric compositional data types. In: MSFP (2012)
[7] Chlipala, A.: Parametric higher-order abstract syntax for mechanized semantics. In: ICFP (2008)

[8] Day, L.E., Hutton, G.: Towards modular compilers for effects. In: TFP (2011)
[9] Day, L.E., Hutton, G.: Compilation à la carte. In: IFL (2013)
[10] Delaware, B., Keuchel, S., Schrijvers, T., Oliveira, B.C.: Modular monadic meta-theory. In: ICFP (2013)
[11] Delaware, B., d. S. Oliveira, B.C., Schrijvers, T.: Meta-theory à la carte. In: POPL (2013)
[12] Ekman, T., Hedin, G.: The jastadd system - modular extensible compiler construction. Sci. Comput. Prog. 69(1-3), 14–26 (2007)
[13] Frisby, N., Kimmell, G., Weaver, P., Alexander, P.: Constructing language processors with algebra combinators. Sci. Comput. Prog. 75(7), 543 – 572 (2010)
[14] Harrison, W.: Modular Compilers and their Correctness Proofs. Ph.D. thesis, University of Illinois at Urbana-Champaign (2001)
[15] Johann, P., Ghani, N.: Foundations for structured programming with GADTs. In: POPL (2008)
[16] Kimmell, G., Komp, E., Alexander, P.: Building compilers by combining algebras. In: ECBS (2005)
[17] Liang, S., Hudak, P., Jones, M.: Monad transformers and modular interpreters. In: POPL. New York, NY, USA (1995)
[18] Marlow, S.: Haskell 2010 language report (2010), `http://www.haskell.org/onlinereport/haskell2010/`
[19] Mernik, M., Umer, V.: Incremental programming language development. Comput. Lang. Syst. Struct. 31(1), 1–16 (2005)
[20] Oliveira, B.C.d.S., Löh, A.: Abstract syntax graphs for domain specific languages. In: PEPM (2013)
[21] Oliveira, B.C., Cook, W.R.: Functional programming with structured graphs. In: ICFP (2012)
[22] Ramsey, N., Dias, J., Peyton Jones, S.: Hoopl: a modular, reusable library for dataflow analysis and transformation. In: Haskell (2010)
[23] Schwaab, C., Siek, J.: Modular type-safety proofs in agda. In: PLPV (2013)
[24] Swierstra, W.: Data types à la carte. J. Funct. Program. 18(4), 423–436 (2008)
[25] Viera, M.: First Class Syntax, Semantics and Their Composition. Ph.D. thesis, Utrecht University (2013)
[26] Viera, M., Swierstra, D.: Attribute grammar macros. In: BSPL (2012)
[27] Viera, M., Swierstra, S.D., Swierstra, W.: Attribute grammars fly first-class. In: ICFP (2009)
[28] Wyk, E.V., Bodin, D., Gao, J., Krishnan, L.: Silver: An extensible attribute grammar system. Sci. Comput. Prog. 75(1-2), 39–54 (2010)
[29] Yakushev, A.R., Holdermans, S., Löh, A., Jeuring, J.: Generic programming with fixed points for mutually recursive datatypes. In: ICFP (2009)