The Architecture of the Utrecht Haskell Compiler

Atze Dijkstra   Jeroen Fokker   S. Doaitse Swierstra

Department of Information and Computing Sciences
Universiteit Utrecht
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
{atze,jeroen,doaitse}@cs.uu.nl

Abstract

In this paper we describe the architecture of the Utrecht Haskell Compiler (UHC). UHC is a new Haskell compiler, that supports most (but not all) Haskell 98 features, plus some experimental extensions. It targets multiple backends, including a bytecode interpreter backend and a full program analysis backend, both via C. The implementation is rigorously organized as stepwise transformations through some explicit intermediate languages. The tree walks of all transformations are expressed as an algebra, with the aid of an Attribute Grammar based preprocessor. The compiler is just one materialization of a framework that supports experimentation with language variants, thanks to an aspect-oriented internal organization.

1. Introduction

On the occasion of the Haskell Hackathon on April 18th, 2009, we announced the first release of a new Haskell compiler: the Utrecht Haskell Compiler, or UHC for short. Until Haskell Prime (Committee 2009) is available as a standard, UHC strives to be a full Haskell 98 (Peyton Jones 2003) compiler (although currently it lacks a few features). The reason that we announce the compiler even though it is not yet fully finished, is that we feel that UHC is mature enough to use for play and experimentation.

One can ask why there is a need for (yet) another Haskell compiler, where the Glasgow Haskell Compiler (GHC) is already available as a widely used, fully featured, production quality Haskell compiler (Marlow et al. 2004; team 1998; Peyton Jones 1996; Peyton Jones and Marlow 2002). In fact, we are using GHC ourselves for the implementation of UHC. Also, various alternatives exist, like Hugs (that in its incarnation of Gofer was the epoch maker for Haskell), and the Haskell compilers from York (NHC/YHC).

Still, we think UHC has something to add to existing compilers, not so much as a production compiler (yet), but more because of its systematically designed and extensible architecture. It is intended to be a platform for those who wish to experiment with adding new language or type system features. In a broader sense, UHC is a framework from which one can construct a series of increasingly complex compilers for languages reaching from simple lambda calculus to (almost-)Haskell 98. The UHC compiler *sensu strictu* is just the culmination point of the series. We have been referring to the framework as 'EHC' (E for extensible, educational, essential, experimental...) in the past (Dijkstra et al. 2007), but for ease we now call both the framework and its main compiler 'UHC'. Internally we use a stepwise and aspect-wise approach, realized by the use of attribute grammars (AG) and other tools.

In its current state, UHC supports most of the Haskell 98 (including polymorphic typing, type classes, input/output, base library), but a few features are still lacking (like defaulting, and some members of the awkward squad (Peyton Jones 2002)). On the other hand, there are some extensions, notably to the type system. The deviations from the standard are not caused by obstinacy or desire to change the standard, but rather because of arbitrary priorization of the feature wish list.

The main structure of the compiler is shown in Figure 1. Haskell source text is translated to an executable program by stepwise transformation. Some of the transformations translate the program to a lower level language, many others are transformations within one language, establishing some invariant or performing some optimization.

All transformations, both within a language and between languages, are expressed as an algebra giving a semantics to the language. The algebras are described with the aid of an attirbute grammar, which makes it possible to write multi-pass tree-traversals without even realizing the exact number of passes. Although the compiler driver is set up to pass data structures between transformations, for all intermediate languages we have a concrete syntax with a parser and a pretty printer. This facilitates debugging the compiler, by inspecting code between transformations.

Here is a short characterization of the intermediate languages. In section 3 we give a more detailed description.

- Haskell (HS): a general-purpose, higher-order, polymorphically typed, lazy functional language.
- Essential Haskell (EH): a higher-order, polymorphically typed, lazy functional language close to lambda-calculus, without syntactic sugar.
- Core: an untyped, lazy functional language close to lambda-calculus (similar to, but not the same as the Core language used in GHC).
- Grin: 'Graph reduction intermediate notation', the instruction set of a virtual machine of a small functional language with strict semantics, with features that enable implementation of laziness (Boquist 1999).
- Silly: 'Simple imperative little language', an abstraction of features found in every imperative language (if-statements, assignments, explicit memory allocation) augmented with primitives

for manipulating a stack, easily translatable to e.g. C (not all features of C are provided, only those that are needed for our purpose).

- BC: A bytecode language for a low-level machine intended to interpret Grin which is not fully program analyzed and transformed. We do not discuss this language in this paper.

The compiler targets different backends, based on a choice of the user. In all cases, the compiler starts compiling on a per module basis, desugaring the Haskell source text to Essential Haskell, type checking it and translating it to Core. Then there is a choice from three modes of operation:

- In *full program analysis mode*, the Core modules of the program and required libraries are assembled together and processed further as a whole. At the Grin level, elaborate inter-module optimization takes place. Ultimately, all functions are translated to low level C, which can be compiled by a standard compiler. As alternative backends, we are experimenting with other target languages, among which are the Common Intermediate Language (CIL) from the Common language infrastructure used by .NET (ISO 2006), and the Low-Level Virtual Machine (LLVM) compiler infrastructure (Lattner and Adve 2004).
- In *bytecode interpreter mode*, the Core modules are translated to Grin separately. Each Grin module is translated into instructions for a custom bytecode machine. The bytecode is emitted in the form of C arrays, which are interpreted by a handwritten bytecode interpreter in C.
- In *Java mode*, the Core modules are translated to bytecode for the Java virtual machine (JVM). Each function is translated to a separate class with an $eval$ function, and each closure is represented by an object combining a function with its parameters. Together with a driver function in Java which steers the interpretation, these can be stored in a Java archive (jar) and be interpreted by a standard Java interpreter.

In Section 2 we describe the tools that play an important role in UHC: the Attribute Grammar preprocessor, a language for expressing type rules, and the variant and aspect manager. In Section 3 we describe the intermediate languages in the UHC pipeline in more detail, together with a running example. In Section 4 the transformations are characterized in more detail. Finally, in Section 5 we draw conclusions about the methodology used, and mention related and future work.

2. Techniques and Tools

2.1 Tree-oriented programming

Using higher order functions on lists, like map, $filter$ and $foldr$, is a good way to abstract from common patterns in functional programs.

The idea that underlies the definition of $foldr$, i.e. to capture the pattern of an inductive definition by having a function parameter for each constructor of the data structure, can also be used for other data types, and even for multiple mutually recursive data types. A function that can be expressed in this way was called a *catamorphism* by Bird, and the collective extra parameters to $foldr$-like functions an *algebra* (Bird 1984; Bird and de Moor 1996). Thus, $((+), 0)$ is an algebra for lists, and $((+\!\!+), [\,])$ is another. In fact, every algebra defines a *semantics* of the data structure. When applying $foldr$-like functions to the algebra consisting of the original constructor functions, such as $((:), [\,])$ for lists, we have the identity function. Such an algebra is said to define the "initial" semantics. Outside circles of functional programmers and category theorists, an algebra is simply known as a "tree walk".

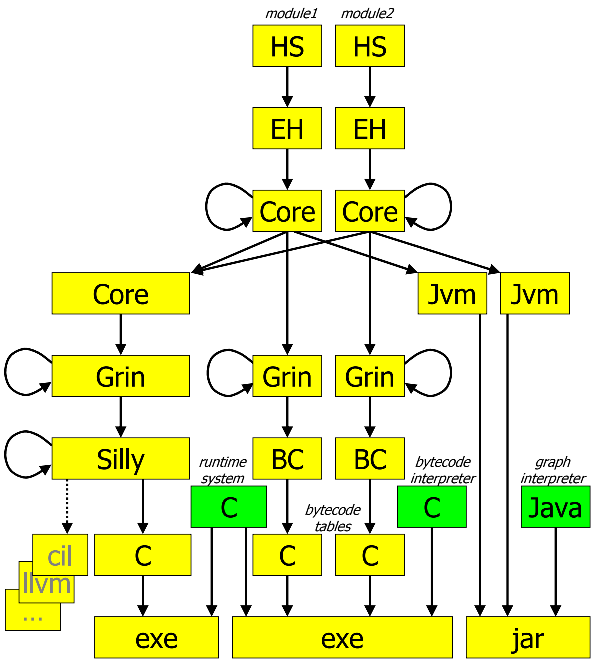

Figure 1. Intermediate languages and transformations in the UHC pipeline, in each of the three operation modes: full program analysis (left), bytecode interpreter (middle), and Java (right).

In compiler construction, algebras are very useful in defining a semantics of a syntactic structure or, bluntly said, to define tree walks over the parse tree. The fact that this is not widely done, is due to the following problems:

1. Unlike lists, for which $foldr$ is standard, in a compiler we deal with custom data structures for abstract syntax of a language, which each need a custom $fold$ function. Moreover, whenever we change the abstract syntax, we need to change the $fold$ function and every algebra.
2. Generated code can be described as a semantics of the language, but often we need additional semantics: listings, messages, and internal structures (symbol tables etc.). This can be done by having the semantic functions in algebras return tuples, but this makes the program hard to maintain.
3. Data structures for abstract syntax tend to have many alternatives, so algebras end up to be clumsy tuples containing dozens of functions.
4. In practice, information not only flows bottom-up in the parse tree, but also top-down. E.g., symbol tables with global definitions need to be distributed to the leafs of the parse tree to be able to evaluate them. This can be done by using higher-order domains for the algebras, but the resulting code becomes even harder to understand.
5. A major portion of the algebra is involved with moving information around. The essence of a semantics usually forms only a small part of the algebra and is obscured by lots of boilerplate.

Some seek the solution to these problems in the use of monads: the reader monad to pass information down into the tree, the writer monad to move information upwards, and the state monad and its derivatives to accumulate information during the tree walk.(Jones 1999).

Despite the attractiveness of staying inside Haskell we think this approach is doomed to fail when the algebras to be described are getting more and more complicated.

To save the nice idea of using an algebra for defining a semantics, we use a preprocessor for Haskell (Swierstra et al. 1999) that overcomes the abovementioned problems. It is not a separate language; we can still use Haskell for writing auxiliary functions, and use all abstraction techniques and libraries available. The preprocessor just allows a few additional constructs, which can be translated into a custom *fold* function and algebras, or an equivalent more efficient implementation. However, if one really wants to avoid a preprocessor, Viera, Swierstra and Swierstra recently described a technique to encode an attribute grammar directly in Haskell while keeping the advantages described below (Viera et al. 2009).

We describe the main features of the preprocessor here, and explain why they overcome the five problems mentioned above. The abstract syntax of the language is defined in a **data** declaration, which is like a Haskell *Data* declaration with named fields, however without the braces and commas. Constructor function names need not to be unique between types. As an example, consider a fragment of a typical imperative language:

```
data Stat
   = Assign dest  :: String   src  :: Expr
   | While  cond  :: Expr     body :: Stat
   | Group  elems :: [Stat]
data Expr
   = Const  num   :: Int
   | Var    name  :: String
   | Add    left  :: Expr     right :: Expr
   | Call   name  :: String   args  :: [Expr]
```

The preprocessor generates corresponding *Data* declarations (making the constructors unique by prepending the type name, like *Expr_Const*), and generates a custom *fold* function. This overcomes problem 1.

For any desired value we wish to compute over a tree, we can declare a "synthesized attribute". Possibly more than one data type can have the same attribute. For example, we can declare that both statements and expressions need to synthesize bytecode as well as listings, and that expressions can be evaluated to integer values:

```
attr Expr Stat syn bytecode :: [Instr]   syn listing :: String
attr Expr      syn value    :: Int
```

The preprocessor generates semantic functions that return tuples of synthesized attributes, but we can simply refer to attributes by name. This overcomes problem 2. Moreover, if at a later stage we add extra attributes, we do not have to refactor a lot of code.

The value of each attribute needs to be defined for every constructor of every data type which has the attribute. Such definitions are known as "semantic rules", and start with keyword **sem**.

```
sem Expr | Const lhs.value = @num
         | Add   lhs.value = @left.value + @right.value
```

This states that the synthesized (left hand side) *value* attribute of a *Const*ant expression is just the contents of the *num* field, and that of an *Add*-expression can be computed by adding the *value* attributes of its subtrees. The @-symbol in this context should be read as "attribute", not to be confused with Haskell "as-patterns". At the left of the =-symbol, the attribute to be defined is mentioned; at the right, the defining Haskell expression is given. The preprocessor collects and orders all definitions in a single algebra, replacing attribute references by suitable selections from the results of the tree walk on the children. This overcomes problem 3.

To be able to pass information downward during a tree walk, we can define "inherited" attributes (the terminology goes back to Knuth (Knuth 1968)). As an example, it can serve to pass down an environment, i.e. a lookup table that associates variables to values, which is needed to evaluate expressions:

```
type Env = [(String, Int)]
attr Expr inh env :: Env
sem Expr | Var lhs.value = fromJust $
                              lookup @lhs.env @name
```

The preprocessor translates inherited attributes into extra parameters for the semantic functions in the algebra. This overcomes problem 4.

In many situations, **sem** rules only specify that attributes a tree node inherits should be passed unchanged to its children, as in a *Reader* monad. To scrap the boilerplate expressing this, the preprocessor has a convention that, unless stated otherwise, attributes with the same name are automatically copied. A similar automated copying is done for synthesized attributes passed up the tree, as in a *Writer* monad. When more than one child offers a synthesized attribute with the required name, we can specify to **use** an operator to combine several candidates:

```
attr Expr Stat syn listing use (++) []
```

which specifies that by default, the synthesized attribute *listing* is the concatenation of the *listing*s of all children that produce a sub-listing, or the empty list if no child produces one. This overcomes problem 5.

2.2 Rule-oriented programming

With the AG language we can describe the part of a compiler related to tree walks concisely and efficiently. However, this does not give us any means of looking at such an implementation in a more formal setting. We use the domain specific language *Ruler* for describing the AG part related to the type system. From a Ruler description we both generate the corresponding AG implementation and a LaTeX rendering for use in text about the formal aspects.

Although the use of Ruler currently is in flux because we are working on a newer version and therefore are only partially using Ruler for type system descriptions, we demonstrate some of its capabilities because it is our intent to tackle the difficulties involved with type system implementations by generating as much as possible automatically from higher level descriptions.

The idea of Ruler is to generate from a single source both generate a *LaTeX* rendering for human use in technical writing:

$$\frac{\begin{array}{c} v \text{ fresh} \\ \Gamma; \mathcal{C}^k; v \to \sigma^k \vdash^e e_1 : \sigma_a \to \sigma \leadsto \mathcal{C}_f \\ \Gamma; \mathcal{C}_f; \sigma_a \vdash^e e_2 : \_ \leadsto \mathcal{C}_a \end{array}}{\Gamma; \mathcal{C}^k; \sigma^k \vdash^e e_1\ e_2 : \mathcal{C}_a \sigma \leadsto \mathcal{C}_a} \text{ E.APP}_{HM}$$

and its corresponding AG implementation:

$$\begin{array}{l}
\textbf{sem}\ \mathit{Expr} \\
\quad \mid \mathit{App}\ (\mathit{func.gUniq}, \textbf{loc}.\mathit{uniq1}) \\
\qquad\qquad\qquad = \mathit{mkNewLevUID}\ @\textbf{lhs}.\mathit{gUniq} \\
\qquad \mathit{func.knTy} = [\mathit{mkTyVar}\ @\mathit{uniq1}]\ \text{`}\mathit{mkArrow}\text{`}\ \ @\textbf{lhs}.\mathit{knTy} \\
\qquad (\textbf{loc}.\mathit{ty\_a\_}, \textbf{loc}.\mathit{ty\_}) \\
\qquad\qquad\qquad = \mathit{tyArrowArgRes}\ @\mathit{func.ty} \\
\qquad \mathit{arg}\ \ .\mathit{knTy} = \ @\mathit{ty\_a\_} \\
\qquad \textbf{loc}\ \ .\mathit{ty}\ \ \ = \ @\mathit{arg.tyVarMp} \oplus \ @\mathit{ty\_}
\end{array}$$

In this paper we neither further discuss the meaning or intention of the above fragments (Dijkstra 2005) nor explain Ruler (Dijkstra and Swierstra 2006) in depth. However, to sketch the underlying ideas we show the Ruler code required for the above output; we need to define the scheme (or type) of a judgment and populate these with actual rules.

A scheme defines a LaTeX output template (`judgeuse tex`) with `holes` to be filled in by rules and a parsing template (`judgespec`).

$$\begin{array}{l}
\textbf{scheme}\ \mathit{expr} = \\
\quad \textbf{holes}\ [\textbf{node}\ e : \mathit{Expr}, \textbf{inh}\ \mathit{valGam} : \mathit{ValGam}, \textbf{inh}\ \mathit{knTy} : \mathit{Ty} \\
\qquad\quad , \textbf{thread}\ \mathit{tyVarMp} : \mathcal{C}, \textbf{syn}\ \mathit{ty} : \mathit{Ty}] \\
\quad \textbf{judgeuse}\ \textbf{tex}\ \mathit{valGam}; \mathit{tyVarMp}.\textbf{inh}; \mathit{knTy} \\
\qquad\qquad\qquad \vdash ..\texttt{"e"}\ e : \mathit{ty} \leadsto \mathit{tyVarMp}.\textbf{syn} \\
\quad \textbf{judgespec}\ \mathit{valGam}; \mathit{tyVarMp}.\textbf{inh}; \mathit{knTy} \\
\qquad\qquad\qquad \vdash e : \mathit{ty} \leadsto \mathit{tyVarMp}.\textbf{syn}
\end{array}$$

The rule for application is then specified by specifying premise judgments (`judge` above the dash) and a conclusion (blow the dash) using the parsing template defined for scheme `expr`.

$$\begin{array}{l}
\textbf{rule}\ e.\mathit{app} = \\
\quad \textbf{judge}\ \mathit{tvarvFresh} \\
\quad \textbf{judge}\ \mathit{expr} = \mathit{tyVarMp}.\textbf{inh}; \mathit{tyVarMp}; (v \rightarrow \mathit{knTy}) \\
\qquad\qquad\qquad \vdash \mathit{eFun} : (\mathit{ty.a} \rightarrow \mathit{ty}) \leadsto \mathit{tyVarMp.fun} \\
\quad \textbf{judge}\ \mathit{expr} = \mathit{tyVarMp.fun}; \mathit{valGam}; \mathit{ty.a} \\
\qquad\qquad\qquad \vdash \mathit{eArg} : \mathit{ty.a} \leadsto \mathit{tyVarMp.arg} \\
\quad - \\
\quad \textbf{judge}\ \mathit{expr} = \mathit{tyVarMp}.\textbf{inh}; \mathit{valGam}; \mathit{knTy} \\
\qquad\qquad\qquad \vdash (\mathit{eFun}\ \mathit{eArg}) \\
\qquad\qquad\qquad : (\mathit{tyVarMp.arg}\ \mathit{ty}) \leadsto \mathit{tyVarMp.arg}
\end{array}$$

For this example no further annotations are required to automatically produce AG code, except for the freshness of a type variable. The judgment `tvarvFresh` encapsulates this by providing the means to insert some handwritten AG code.

In summary, the basic idea of Ruler thus is to provide a description resembling the original type rule as much as possible, and then helping the system with annotations to allow the generation of an implementation and a LaTeX rendering. Obviously the above examples does not contain all the annotations required to achieve this.

2.3 Aspect-oriented programming

UHC's source code is organized into small fragments, each belonging to a particular *variant* and *aspect*. A variant represents a step in a sequence of languages, where each step adds some language features, starting with simply typed lambda calculus and ending with UHC. Each step builds on top of the previous one. Independent of a variant each step adds features in terms of aspects. For example, the type system and code generation are defined as different aspects. UHC's build system allows for selectively building a compiler for a variant and a set of aspects.

Source code fragments assigned to a variant and aspects are stored in *chunked* text files. A tool called *Shuffle* then generates the actual source code when parameterized with the desired variant and aspects. Shuffle is language neutral, so all varieties of implementation languages can be stored in chunked format. For example, the following chunk defines a Haskell wrapper for variant 2 for the construction of a type variable:

```
%%[(2 hmtyinfer || hmtyast).mkTyVar
mkTyVar :: TyVarId -> Ty
mkTyVar tv = Ty_Var tv
%%]
```

The notation `%%[(2 hmtyinfer | hmtyast).mkTyVar` begins a chunk for variant 2 with name $\mathit{mkTyVar}$ for aspect `hmtyinfer` (Hindley-Milner type inference) or `hmtyast` (Hindley-Milner type abstract syntax), ended by `%%]`. Processing by Shuffle then gives:

$$\begin{array}{l}
\mathit{mkTyVar} :: \mathit{TyVarId} \rightarrow \mathit{Ty} \\
\mathit{mkTyVar}\ \mathit{tv} = \mathit{Ty\_Var}\ \mathit{tv}
\end{array}$$

The subsequent variant 3 requires a more elaborate encoding of a type variable (we do not discuss this further). The wrapper must be redefined, which we achieve by explicitly overriding `2.mkTyVar` by a chunk for `3.mkTyVar`:

```
%%[(3 hmtyinfer || hmtyast).mkTyVar -2.mkTyVar
mkTyVar :: TyVarId -> Ty
mkTyVar tv = Ty_Var tv TyVarCateg_Plain
%%]
```

Although the type signature can be factored out, we refrain from doing so for small definitions.

Chunked sources are organized on a per file basis. Each chunked file for source code for UHC is processed by Shuffle to yield a corresponding file for further processing, depending on the language used. For chunked Haskell a single module is generated, for chunked AG the file may be combined with other AG files by the AG compiler.

The AG compiler itself also supports a notion of aspects, different from Shuffle's idea of aspects in that it allows definitions for attributes and abstract syntax to be defined independent of file and position in a file. Attribute definitions and attribute equations thus can be grouped according to the programmers sense of what should be together; the AG compiler combines all these definitions and generates corresponding Haskell code.

Finally, chunked files may be combined by Shuffle by means of explicit reference to the name of a chunk. This also gives a form of literate programming literate programming tools (Knuth 1984) where text is generated by explicitly combining smaller text chunks. For example, the above code for `2.mkTyVar` and `3.mkTyVar` are extracted from the chunked source code of UHC and combined with the text for this explanation by Shuffle.

3. Languages

The compiler translates a Haskell program to executable code by applying many small transformations. In the process, the program is represented using five different data structures, or languages. Some transformations map one of these languages to the next, some are transformations within one language. Together, the five languages span the spectrum between a full feature, lazy functional language, and a limited, low level imperative language.

3.1 The Haskell Language

The Haskell language (HS) closely follows Haskell's concrete syntax. A combinator-based, error-correcting parser parses the source text and generates a HS parse tree. It consists of numerous datatypes, some of which have many constructors. A *Module* consists of a name, exports, and declarations. Declarations can be varied: function bindings, pattern bindings, type signatures, data types, new types, type synonyms, class, instance. . . Function bindings involve a right hand side which is either an expression or a list of guarded expressions. An expression, in turn, has no less than 29 alternatives. All in all, the description of the context-free grammar consists of about 1000 lines of code.

We maintain sufficient information in the abstract syntax tree to reconstruct the original input; this includes parentheses, source code position, etc. (but no comments and whitespace).

When processing HS we deal with the following tasks:

- *Name resolution:* Checking for properly introduced names and renaming all identifiers to the equivalent fully qualified name.
- *Operator fixity and precedence:* Expressions are parsed without taking into account the fixity and precedence of operators. Expressions are rewritten to remedy this.
- *Name dependency:* Definitions are reordered into different let bindings such that all identifier uses come after their definition. Mutually recursive definitions are put into one let binding.
- *Definition gathering:* Multiple definitions for the same identifier are merged into one.
- *Desugaring:* List comprehension, **do**-notation, etc. are desugared.

In the remainder of this section on languages we use the following running example program to show how the various intermediate languages are used:

```
module M where
len :: [a] → Int
len [] = 0
len (x : xs) = 1 + len xs
main = putStr (show (len (replicate 4 'x')))
```

3.2 The Essential Haskell Language

HS processing generates Essential Haskell (EH). The EH equivalent of the running example is shown below. Some details have been omitted and replaced by dots.

```
let  M.len :: [a] → Int
     M.len
        = λx1 → case x1 of
          UHC.Prelude.[]
             → UHC.Prelude.fromInteger  0
          (UHC.Prelude.: x xs )
             → ...
in
let M.main =  UHC.Prelude.putStr ...
in
let main ::  UHC.Prelude.IO ...
    main = UHC.Prelude.ehcRunMain  M.main
in
main
```

In constrast to the HS language, the EH language brings back the language to its essence, removing as much syntactic sugar as is possible. An EH module consists of a single expression only, which is the body of the *main* function, with local let-bindings for the other top-level values.

Processing EH deals with the following tasks:

- *Type system:* Type analysis is done, types are erased when Core is generated.
- *Evaluation:* Enforcing evaluation is made explicit by means of a **let**! Core construct.
- *Recursion:* Recursiveness is made explicit by means of a **let** *rec* Core construct.
- *Type classes:* All evidence for type class predicates are transformed to explicit dictionary parameters.
- *Patterns:* Patterns are transformed to their more basic equivalent, inspecting one constructor at a time, etc. .

3.3 The Core Language

The Core language is basically the same as lambda-calculus. The Core equivalent of the running example program is:

```
module M =
letrec
  {M.len =
     λM.x1_1 →
       let !{_2 = M.x1_1} in
       case _2 of
         { C : {..., ...} → ...
         ; C[]{} →
           let
             {_3 =
                (UHC.Prelude.packedStringToInteger)
                  (#String "0")} in
           let
             {_4 =
                (UHC.Prelude.fromInteger)
                  (UHC.Prelude._d1_Num : DICT )
                  (_3)} in
           _4
         }
in ...
```

A Core module, apart from its name, consists of nothing more than an expression, which can be thought of as the body of *main*:

```
data CModule
   = Mod nm :: Name expr :: CExpr
```

An expression resembles an expression in lambda calculus. We have constants, variables, and lambda abstractions and applications of one argument:

```
data CExpr
   = Int     int :: Int
   | Char    char :: Char
   | String  str :: String
   | Var     name :: Name
   | Tup     tag :: Tag
   | Lam     arg :: Name       body :: CExpr
   | App     func :: CExpr  arg :: Cexpr
```

Alternative *Tup* encodes a constructor, to be used with *App* to construct actual data alternatives or tuples. The *Tag* of a *Tup* encodes the *Int* tag, arity, and other information.

Furthermore, there is case distinction and local binding:

```
| Case expr :: CExpr  alts :: [CAlt]       dflt :: CExpr
| Let   categ :: Categ  binds :: [CBind]  body :: CExpr
```

The *categ* of a *Let* describes whether the binding is recursive, strict, or plain. These two constructs use the auxiliary notions of alternative and binding:

data $CAlt$
$\quad = Alt \quad pat : CPat \quad expr :: CExpr$
data $CBind$
$\quad = Bind\ name : Name\ expr :: CExpr$
$\quad \mid FFI \quad name : Name\ imp :: String\ ty :: Ty$

A pattern introduces bindings, either directly or as a field of a constructor:

data $CPat$
$\quad = Var \quad name :: Name$
$\quad \mid Con \quad name :: Name\ tag :: Tag\ binds :: [CPatBind]$
$\quad \mid BoolExpr\ name :: Name\ cexpr :: CExpr$
data $CPatBind$
$\quad = Bind \quad offset :: Int \quad pat :: CPat$

The actual Core language is more complex because of the following:

- Experiments with extensible records; we omit this part as extensible records are currently not supported in UHC.
- Type classes involve code generation because only after context reduction dictionary values and coercions are known. Core has syntax for expressing holes to be filled in later; this is a mechanism similar to type variables representing yet unknown types.
- An annotation mechanism is used to propagate information about dictionary values. This mechanism is somewhat ad hoc and we expect it to be changed when more analyses are done in earlier stages of the compiler.

3.4 The Grin Language

The Grin equivalent of the running example program is:

```
module M
{M.len M.x_1_1 =
  {eval M.x_1_1; λ_2 →
    case _2 of
      {C /: →
        {...}
      ; C / [] →
        {store (C/UHC.Prelude.PackedString "0"); λ_6 →
         store (F/UHC.Prelude.packedStringToInteger _6);
            λ_3 →
         store (P/0/UHC.Prelude.fromInteger
                    UHC.Prelude._d_1_Num); λ_5 →
         store (A /apply _5 _3); λ_4 →
         eval  _4 }
      }}}
```

A Grin module consists of its name, global variables with their initializations, and bindings of function names with parameters to their bodies.

data $GrModule$
$\quad = Mod\ \ nm :: Name\ globals :: [GrGlobal]\ binds :: [GrBind]$
data $GrGlobal$
$\quad = Glob\ \ nm :: Name\ val :: GrVal$
data $GrBind$
$\quad = Bind\ nm :: Name\ args :: [Name]\ body :: GrExpr$

Values manipulated in the Grin language are varied: we have nodes (think: heap records) consisting of a tag and a list of fields, standalone tags, basic ints and strings, pointers to nodes, and 'empty'. Some of these are directly representable in the languages (nodes, tags, ints and strings)

data $GrVal$
$\quad = LitInt\ int :: Int$
$\quad \mid LitStr\ str :: String$
$\quad \mid Tag \quad tag :: GrTag$
$\quad \mid Node \quad tag :: GrTag\ flds :: [GrVal]$

Pointers to nodes are also values, but they have no direct denotation. On the other hand, variables ranging over values are not a value themselves, bur for syntactical convenience we do add the notion of a 'variable' to the $GrVal$ data type:

$\mid Var\ name :: Name$

The tag of a node describes its role. It can be a constructor of a datatype (Con), a function of which the call is deferred because of lazy evaluation (Fun), a function that is partially parameterized but still needs more arguments ($PApp$), or a deferred application *apply* of an unknown function (appearing as the first argument of the node) to a list arguments (App).

data $GrTag$
$\quad = Con \quad name :: Name$
$\quad \mid Fun \quad name :: Name$
$\quad \mid PApp\ needs :: Int\ name :: Name$
$\quad \mid App \quad applyfn :: Name$

The four tags are represented as C, F, P and A in the example program above.

The body of a function denotes the calculation of a value, which is represented in a program by an 'expression'. Expressions can be combined in a monadic style. Thus we have $Unit$ for describing a computation immediately returning a value, and Seq for binding a computation to a variable (or rather a lambda pattern), to be used subsequently in another computation:

data $GrExpr$
$\quad = Unit\ val :: GrVal$
$\quad \mid Seq \quad expr :: GrExpr\ pat :: GrPatLam\ body :: GrExpr$

There are some primitive computations (that is, constants in the monad) one for storing a node value (returning a pointer value), and two for fetching a node previously stored, and for fetching one field thereof:

$\mid Store \quad val :: GrVal$
$\mid FetchNode\ name :: Name$
$\mid FetchField\ name :: Name\ offset :: Int$

Other primitive computations call Grin and foreign functions, respectively. The name mentioned is that of a known function (i.e., there are no function variables) and the argument list should fully saturate it:

$\mid Call\ name :: Name\ \ args :: [GrVal]$
$\mid FFI\ \ name :: String\ args :: [GrVal]$

Two special primitive computations are provided for evaluating node that may contain a Fun tag, and for applying a node that must contain a $PApp$ tag (a partially parameterized function) to further arguments:

$\mid Eval\ name :: Name$
$\mid App\ \ name :: Name\ args :: [GrVal]$

Next, there is a computation for selecting a matching alternative, given the name of the variabele containing a node pointer:

$\mid Case\ val :: GrVal\ alts :: [GrAlt]$

Finally, we need a primitive computation to express the need of 'updating' a variable after it is evaluated. Boquist proposed an $Update$ expression for the purpose which has a side effect only and an ‘empty’ result value (Boquist 1999). We observed that the need for updates is always next to either a *FetchNode* or a *Unit*, and found it more practical and more efficient to introduce two update primitives:

$\mid$ *FetchUpdate* *src* :: *Name* *dst* :: *Name*
$\mid$ *UpdateUnit* *name* :: *Name* *val* :: *GrVal*

Auxiliary data structures are that for describing a single alternative in a *Case* expression:

data *GrAlt*
  $\mid$ *Alt* *pat* :: *GrPatAlt* *expr* :: *GrExpr*

And for two kinds of patterns, occurring in a *Seq* expression and in an *Alt* alternative, respectively. A simplified version of these is the following, but we will need some more constructors for patterns later.

data *GrPatLam*
  = *Var* *name* :: *Name*
data *GrPatAlt*
  = *Node* *tag* :: *GrTag* *args* :: [*Name*]

4. Transformations

4.1 Core Transformations

Three major gaps have to be bridged in the transformation from Core to Grin. Firstly, where Core has a lazy semantics, in Grin deferring of function calls and their later evaluation is explicitly encoded. Secondly, in Core we can have local function definitions, whereas in Grin all function definitions are at top level. Grin does have a mechanism for local, explicitly sequenced variable bindings. Thirdly, whereas Core functions always have one argument, in Grin functions can have multiple parameters, but they take them all at the same time. Therefore a mechanism for partial parametrization is necessary. The end result is lambda lifted Core, that is the floating of lambda-expressions to the top level and passing of non-global variables explicitly as parameters.

The Core transformations listed below also perform some trivial cleanup and optimizations, because we avoid burdening the Core generation from EH with such aspects.

1. *EtaReduction* Perform η-reduction, that is replace expressions like $\lambda x\ y \rightarrow f\ x\ y$ with f. Such expressions are introduced by coercions which (after context reduction) turn out not to coerce anything at all.
2. *RenameUnique* Renames variables such that all variables are globally unique.
3. *LetUnrec* Replace mutually recursive bindings

   $\mathbf{let}\ rec\{v_1 = ..; v_2 = ..\}\ \mathbf{in}\, ..$

   which actually are not mutually recursive by plain bindings

   $\mathbf{let}\ v_1 = ..\ \mathbf{in}\ \mathbf{let}\ v_2 = ..\ \mathbf{in}\, ..$

   Such bindings are introduced because some bindings are potentially mutually recursive, in particular groups of dictionaries.
4. *InlineLetAlias* Inlines let bindings for variables and constants.
5. *ElimTrivApp* Eliminates application of the id function.
6. *ConstProp* Performs addition of int constants at compile time.
7. *FullLazy* Complex expressions like

   $f\ (g\ a)\ (h\ b)$

   are broken up into a sequence of bindings and simpler expressions

   $\mathbf{let}\ v_1 = g\ a\ \mathbf{in}\ \mathbf{let}\ v_2 = h\ b\ \mathbf{in}\ f\ v_1\ v_2$

   which only have variable references as their subexpressions.
8. *LamGlobalAsArg* Pass global variables of let-bound lambda-expressions as explicit parameters, as a preparation for lambda-lifting.
9. *CAFGlobalAsArg* Similar for let-bound constant applicative forms (CAFs).
10. *FloatToGlobal* Performs ’lambda lifting’: move bindings of lambda-expressions and CAFs to the global level.
11. *LiftDictFields* Makes sure that all dictionary fields exist as a top-level binding.
12. *FindNullaries* Finds nullary (parameterless) functions and duplicates them; the two copies are differently annotated, such that one of the two will end up as an updateable global variable.

After the transformations, translation to Grin is performed, where the following issues are addressed:

- for *Let*-expressions: global expressions are collected and made into Grin function bindings; local non-recursive expressions are sequenced by Grin *Seq*-expressions; for local recursive let-bindings a *Seq*uence is created which starts out to bind a new variable to a ‘black hole’ node, then processes the body, and finally generates a *FetchUpdate*-expression for the introduced variable.
- for *Case*-expressions: an explicit *Eval*-expression for the scrutinee is generated, in *Seq*uence with a Grin *Case*-expression.
- for *App*-expressions: it is determined what it is that is applied:
  - if it is a constructor, then a node with *Con* tag is returned;
  - if it is a lambda of known arity which has exactly the right number of arguments, then either a *Call*-expression is generated (in strict contexts) or a node with *Fun* tag is stored with a *Store*-expression (in lazy contexts);
  - if it is a lambda of known arity that is undersaturated (has not enough arguments), then a node with *PApp* tag is returned (in strict contexts) or *Store*d (in lazy contexts)
  - if it is a lambda of known arity that is oversaturated (has too many arguments), then (in strict contexts) first a *Call*-expression to the function is generated that applies the function to some of the arguments, and the result is bound to a variable that is sub*Seq*uently *App*lied to the remaining arguments; or (in non-strict contexts) a node with *Fun* tag is *Store*d, and bound to a variable that is used in another node which has an *App* tag.
  - if it is a variable that represents a function of unknown arity, then (in strict contexts) the variable is explicitly *Eval*uated, and its result used in an *App*expression to the arguments; or (in non-strict contexts) as a last resort, both function variable and arguments are stored in a node with *App* tag.
- for global bindings: lambda abstractions are ‘peeled off’ the body, to become the arguments of a Grin function binding.
- for foreign function bindings: functions with *IO* result type are treated specially.

We have now reached the point in the compilation pipeline where we perform our full-program analysis. The Core module of the program under compilation is merged with the Core modules of all used libraries. The resulting big Core module is then translated to Grin.

4.2 Grin Transformations

In the Grin world, we take the opportunity to perform many optimizing transformations. Other transformations are designed to move from graph manipulation concepts (complete nodes that can be 'fetched', 'evaluated' and pattern matched for) to a lower level where single word values are moved and inspected in the imperative target language.

We first list all transformations in the order they are performed, and then discuss some issues that are tackled with the combined effort of multiple transformations.

1. *DropUnreachableBindings* Drops all functions not reachable from $main$, either through direct calls, or through nodes that store a deferred or partially parameterized function. The transformation performs a provisional numbering of all functions, and creates a graph of dependencies. A standard reachability algorithm determines which functions are reachable from $main$; the others are dropped.
2. *MergeInstance* Introduces an explicit dictionary for each instance declaration, by merging the default definitions of functions taken from class declarations. This is possible because we have the full program available now (see discussion below).
3. *MemberSelect* Looks for the selection of a function from a dictionary and its subsequent application to parameters. Replaces that by a direct call.
4. *DropUnreachableBindings* Drops the now obsolete implicit constructions of dictionaries.
5. *Cleanup* Replaces some node tags by equivalent ones: $PApp$ 0, a partial application needing 0 more parameters, is changed into Fun, a simple deferred function; deferred applications of constructor functions are changed to immediate application of the constructor function.
6. *SimpleNullary* Optimises nullary functions that immediately return a value or call another function, by inlining them in nodes that encode their deferred application.
7. *ConstInt* Replaces deferred applications of $integer2int$ to constant integers by a constant int. This situation occurs for every numeric literal in the source program, because of the way literals are overloaded in Haskell.
8. *BuildAppBindings* Introduces bindings for $apply$ functions with as many parameters as are needed in the program.
9. *GlobalConstants* Introduces global variables for each constant found in the program, instead of allocation the constants locally.
10. *Inline* Inlines functions that are used only once at their call site.
11. *SingleCase* Replaces case expressions that have a single alternative by the body of that alternative.
12. *EvalStored* Do not do $Eval$ on pointers that bind the result of a previous $Store$. Instead, do a $Call$ if the stored node is a deferred call (with a Fun tag), or do a $Unit$ of the stored node for other nodes.
13. *ApplyUnited* Do not perform $Apply$ on variables that bind the result of a previous $Unit$ of a node with a $PApp$ tag. Instead, do a $Call$ of the function if it is now saturated, or build a new $PApp$ node if it is undersaturated.
14. *SpecConst* Specialize functions that are called with a constant argument. The transformation is useful for creating a specialized 'increment' function instead of $plus$ 1, but its main merit lies in making specialized versions of overloaded functions, that is functions that take a dictionary argument. If the dictionary is a constant, specialization exposes new opportunities for the *MemberSelect* transformation, which is why *SpecConst* is iterated in conjunction with *EvalStored*, *ApplyUnited* and *MemberSelect*.
15. *DropUnreachableBindings* Drops unspecialized functions that may have become obsolete.
16. *NumberIdents* Attaches an unique number to each variable and function name.
17. *HeapPointsTo* Does a 'heap points to analysis' (HPT), which is an abstract interpretation of the program in order to determine the possible tags of the nodes that each variable can refer to.
18. *InlineEA* Replaces all occurrences of $Eval$ and $Apply$ to equivalent constructs. Each $Eval\ x$ is replaced by $FetchNode\ x$, followed by a $Case$ distinction on all possible tag values of the node referred to by x, which was revealed by the HPT analysis. If the number of cases is prohibitively large, we resort to a $Call$ to a generic $evaluate$ function, that is generated for the purpose and that distinguishes all possible node tags. Each $App\ f\ x$ construct, that is used to apply an unknown function f to argument x, is replaced by a $Case$ distinction on all possible $PApp$ tag values of the node referred to by f.
19. *ImpossibleCase* Removes alternatives from $Case$ constructs that, according to the HPT analysis, can never occur.
20. *LateInline* Inlines functions that are used only once at their call site. New opportunities for this transformation are present because the *InlineEA* transformation introduces new $Call$ constructs.
21. *SingleCase* Replaces case expressions that have a single alternative by the body of that alternative. New opportunities for this transformation are present because the *InlineEA* transformation introduces new $Case$ constructs.
22. *DropUnusedExpr* Removes bindings to variables if the variable is never used, but only when the expression has no side effect. Therefore, an analysis is done to determine which expressions may have side effects. $Update$ and FFI expressions are assumed to have side effect, and $Case$ and Seq expressions if one of their children has them. The tricky one is $Call$, which has a side effect if its body does. This is circular definition of 'has a side effect' if the function is recursive. Thus we take a 2-pass approach: a 'coarse' approximation that assumes that every $Call$ has a side effect, and a 'fine' approximation that takes into account the coarse approximation for the body. Variables that are never used but which are retained because of the possible side effects of their bodies are replaced by wildcards.
23. *MergeCase* Merges two adjacent $Case$ constructs into a single one in some situations.
24. *LowerGrin* Translates to a lower level version of Grin, in which variables never represent a node. Instead, variables are introduced for the separate fields, of which the number became known through HPT analysis. Also, after this transformation $Case$ constructs scrutinize on tags rather than full nodes.
25. *CopyPropagation* Shortcuts repeated copying of variables.
26. *SplitFetch* Translates to an even lower level version of Grin, in which the node referred to by a pointer is not fetched as a whole, but field by field. That is, the $FetchNode$ expression is replaced by a series of $FetchField$ expressions. The first of these fetches the tag, the others are specialized in the alternatives of the $Case$ expression that always follows a $FetchNode$ expression, such that no more fields are fetched than required by this particular tag.

27. *DropUnusedExpr* Removes variable bindings introduced by *LowerGrin* if they happen not to be used.

28. *CopyPropagation* Again shortcuts repeated copying of variables.

Simplification The Grin language features constructs for manipulating heap nodes, including ones that encode deferred function calls, that are explicitly triggered by an $Eval$ expression. As part of the simplification, this high level construct should be decomposed in smaller steps. Two strategies can be taken to implement evaluation:

- *tagged*: nodes are tagged by small numbers, evaluation is performed by calling a special $evaluate$ function that scrutinizes the tag, and for each possible Fun tag calls the corresponding function and updates the thunk;
- *tagless*: nodes are tagged by pointers to code that does the call and update operations, thus evaluation is tantamount to just jumping to the code pointed to by the tag.

The tagged approach has overhead in calling $evaluate$, but the tagless approach has the disadvantage that the indirect jump involved may stall the lookahead buffer of pipelined processors. Boquist proposed to inline the $evaluate$ function at every occurrence of $Eval$, where for every instance the $Case$ expression involved only contains those cases which can actually occur. It is this approach that we take in UHC.

This way, they high level concept of $Eval$ is replaced by lower level concepts of $FetchNode$, $Case$, $Call$ and $Update$. In turn, each $FetchNode$ expression is replaced by a series of $FetchField$ expressions in a later transformation, and the $Case$ that scrutinizes a node is replaced by one that scrutinizes the tag only.

Abstract interpretation The desire to inline a specialized version of $evaluate$ at every $Eval$ instance brings the need for an analysis that for each pointer variable determines the possible tags of the node. An abstract interpretation of the program, known as 'heap points to (HPT) analysis' tries to approximate this knowledge. As a preparation, the program is scanned to collect constraints on variables. Some constraints immediately provides the information needed (e.g., the variable that binds the result of a $Store$ expression is obviously a pointer to a node with the tag of the node that was stored), but other constraints are indirect (e.g., the variable that binds the result of a $Call$ expression will have the same value as the called function returns). The analysis is essentially a full-program analysis, as actual parameters of functions impose constraints on the parameters.

The constraint set is solved in a fixpoint iteration, which processes the indirect constraints based on information gathered thus far. In order to have fast access to the mapping that records the abstract value for each variable, we uniquely number all variables, and use mutable arrays to store the mapping.

Special attention deserves the processing of the constraint that expresses that x binds the result of $Eval\ p$. If p is already known to point to nodes with a Con tag (i.e., values) then this is also a possible value for x. If p is known to point to nodes with a $Fun\ f$ tag (i.e., deferred functions), then the possible results for f are also possible values for x. And if p is known to point to nodes with an $App\ apply$ tag (i.e., generic applications of unknown functions by $apply$), then the possible results for $apply$ are also possible values for x. For a more detailed description of the algorithm, we refer to another paper (Fokker and Swierstra 2008).

HPT performance The HPT analysis must at least find all possible tags for each pointer, but it is sound if it reports a superset of these. The design of the HPT analysis is a tradeoff between time (the number of iterations it takes to find the fixed point) and accuracy. A trivial solution is to report (in 1 step) that every pointer may point to every tag; a perfect solution would solve the halting problem and thus would take infinite time in some situations.

We found that the number of iterations our implementation takes, is dependent of two factors: the depth of the call graph (usually bounded by a dozen or so in practice), and the length of static data structures in the program. The latter surprised us, but is understandable if one considers the program

$$main = putStrLn\ (show\ (last\ [id, id, id, id, succ]\ 1))$$

where it takes 5 iterations to find out that 1 is a possible parameter of $succ$.

As for accuracy, our HPT algorithm works well for first-order functions. In the presence of many higher-order functions, the results suffer from 'pollution': the use of a higher-order function in one context also influences its result in another context. We counter this undesired behavior in a number of ways:

- instead of using a generic $apply$ function, the *BuildAppBindings* transformation makes a fresh copy for each use by an App tag. This prevents mutual pollution of $apply$ results, and also increases the probability that the $apply$ function can be inlined later;
- we specialize overloaded functions for every dictionary that it is used with, to avoid the $Apply$ needed on the unknown function taken from the dictionary;
- we fall back on explicitly calling $evaluate$ (instead of inlining it) in situations where the number of possible tags is unreasonable large.

Instance declarations The basic idea of implementing instances is simple: an instance is a tuple (known as a 'dictionary') containing all member functions, which is passed as an additional parameter to overloaded functions. Things are complicated, however, by the presence of default implementations in classes: the dictionary for an instance declaration is a merge of the default implementations and the implementations in the instance declaration. Even worse, the class declaration may reside in another module than the instance declaration, and still be mutually dependent with it. Think of the Eq class, having mutually circular definitions of eq and ne, leaving it to the instance declaration to implement either one of them (or both).

A clever scheme was designed by Faxén to generate the dictionary from a generator function that is parameterized by the dictionary containing the default implementations, while the default dictionary is generated from a generator function parameterized by the instance dictionary (Faxén 2002). Lazy evaluation and black holes make this all work, and we employ this scheme in UHC too. It would be a waste, however, now that we are in a full program analysis situation, not to try to do as much work as possible at compile time.

Firstly, we have to merge the default and instance dictionaries. In the Grin world, we have to deal with what the Core2Grin transformation makes of the Faxén scheme. That is:

- A 1-ary generator function gfd that, given a default dictionary, will generate the dictionary;
- A 0-ary function fd that binds a variable to a black hole, calls gfd, and returns the result
- A global variable d which is bound to a node with tag $Fun\ fd$.

We want to change this in a situation where d is bound directly to the dictionary node. This involves reverse engineering the definitions of d, fd and gfd to find the actual member function names buried deep in the definition of gfd. Although possible, this is very volatile as it depends on the details of the Core2Grin translation. Instead, we take a different approach: the definition of fd is annotated with the names of the member functions at the time when it is still explicitly available, that is during the EH2Core translation. Similarly, class definitions are annotated with the names of the default functions. Now the *Grin.MergeInstance* transformation can easily collect the required dictionary fields, provided that the *Core.LiftDictFields* transformation ensures they are available as top-level functions. The fd and gfd functions are obsolete afterwards, and can be discarded by a later reachability analysis.

Secondly, we hunt the program for dictionaries d (as constructed above) and selection functions s_k (easily recognizable as a function that pattern-matches its parameter to a dictionary structure and returns its kth field x_k). In such situations $Call\ \ s_k\ \ d$ can be replaced by $Eval\ x_k$. A deferred member selection, involving a node with tag $Fun\ s_k$ and field d, is dealt with similarly: both are done by the *MemberSelect* transformation.

Thirdly, as x_k is a dictionary field, it is a known node n. If n has a $Fun\ f$ tag, then $Eval\ x_k$ can be replaced by $Call\ f$, and otherwise it can be replaced by $Unit\ n$. This is done by the *EvalStored* transformation. The new $Unit$ that is exposed by this transformation can be combined with the App expression that idiomatically follows the member selection, which is what *ApplyUnited* does.

All of this only works when members are selected from a constant dictionary. Overloaded functions however operate on dictionaries that are passed as parameter, and member selection from a variable dictionary is not caught by *MemberSelect*. The constant dictionary appears where the overloaded function is called, and can be brought to the position where it is needed by specializing functions when they are called with constant arguments. This is done in the *SpecConst* transformation. Besides useful in the chain of transformations that together remove the dictionaries, it is useful for removal of other constants, like a 1-ary successor function as a specialization of $plus\ \ 1$. (If constant specialization is also done for string constants, we get many specializations of $putStrLn$).

The whole pack of transformations is applied repeatedly, as applying them exposes new opportunities for sub-dictionaries. Four iterations suffice to deal with the common cases (involving Eq, Ord, $Integral$, $Read$ etc.) from the prelude.

The only situation where dictionaries cannot be eliminated completely, is where an infinite family of dictionaries is necessary, such as arises from the $Eq\ \ a\ \Rightarrow\ Eq\ \ [a]$ instance declaration in the prelude. In this situation we automatically fall back to the Faxén scheme.

4.3 Silly Transformations

1. *InlineExpr* Avoids copying variables to other variables, if in all uses the original one could be used just as well (i.e., are not modified in between).
2. *ElimUnused* Eliminates assignments to variables that are never used.
3. *EmbedVars* Silly has a notion of function arguments and local variables. After this transformation, these kind of variables are not used anymore, but replaced by explicit stack offsets. So, this transformation does the mapping of variables to stack positions (and, if available, registers). In a tail call, the parameters of the function that is called overwrites the parameters and local variables of the function that does the call. The assignments are scheduled in such a way that no values are overridden that are still needed in assignments to follow.
4. *GroupAllocs* This transformation combines separate, adjacent calls to $malloc$ into one, enabling to do heap overflow check only once for all the memory that is allocated in a particular function.

subsystem	All variants and aspects				UHC only	
	AG	HS	C	total	total	fract.
utility/general	1.7	18.3		20.0	14.0	70%
Haskell	6.7	3.3		9.9	6.9	70%
EH	11.2	0.6		11.8	6.7	57%
EH typing	8.0	7.5		15.5	7.0	45%
Core	7.1	1.0		8.0	4.7	58%
ByteCode	2.1			2.1	1.7	82%
Grin	11.3	1.6		12.9	8.5	66%
Silly	2.8			2.8	2.6	93%
exp.backends	2.5	0.4		2.9	0.8	26%
runtime system			8.1	8.1	6.2	77%
garb.collector			6.0	6.0	0.7	11%
total	53.4	32.5	14.1	100.0	59.8	60%

Figure 2. Code size (in 1000 lines of code) of source files containing Attribute Grammar code (AG), Haskell code (HS) and C code (C), for various subsystems. Column 'all variants' is the total code base for all variants and aspects, column 'UHC' is the selection of the standard compiler, where 'fract.' shows the fraction of the full code base that is selected for UHC.

5. Conclusion

5.1 Code size

UHC is the standard materialization of a more general code base (the UHC framework, formerly known as EHC), from which increasingly powerful 'variants' of the compiler can be drawn, where independent experimental 'aspects' can be switched on or off. The whole source code base consists of a fairly exact 100.000 lines of code. Just over half of it is Attribute Grammar code, which of course has lots of embedded Haskell code in it. One third of the code base is plain Haskell (mostly for utility functions, the compiler driver, and the type inferencer), and one sixth is C (for the runtime system and a garbage collector).

In Figure 2 the breakdown of code size over various subsystems in the pipeline is shown. All numbers are in kilo-lines-of-code, but because of the total of 100.000 lines they can also be interpreted as percentages. Column 'UHC only' shows the size of the code that is selected by Shuffle for the standard compiler, i.e. the most powerful variant without experimental aspects. On average, 60% of the total code base is used in UHC. The rest is either code for low variants which is overwritten in higher variants, code for experimental aspects that are switched off in UHC, chunk header overhead, or comments that were placed outside chunks.

The fraction of code used for UHC is relatively low in the type inferencer (as there are many experimental aspects here), in the experimental backends like Java, Cil and LLVM (as most of them are switched off), and in the garbage collector (as it is not yet used: UHC by default uses the Boehm garbage collector (Boehm and Weiser 1988; Boehm 2006)).

5.2 Methodological observations

Internal organization UHC and its framework use an aspect-wise organization, in which as much as possible is described by higher level domain specific languages from which we generate lower level implementations. UHC as a framework offers a set of compilers, thus allowing picking and choosing a starting point for play and experimentation. This makes UHC a good starting point for research, but debugging is also facilitated by it. A problem can more easily be pinpointed to originate in a particular step of the whole sequence of language increments; the framework then allows to debug the compiler in this limited context, with less interaction by other features.

The stepwise organization, where language features are built on top of each other, offers a degree of isolation. Much better would be to completely independently describe language features. However, this is hard to accomplish because language features always interact and require redefinition of parts of their independent implementation when combined. To do this for arbitrary combinations would be much more complicated then to do it for a sequence of increments.

AG Design Patterns We tend to use various AG idioms frequently. For example, information often is gathered over a tree via a synthesized attribute, subsequently to be passed back as an inherited attribute. This leads to a 2-pass tree traversal.

Some idiomatic use is directly supported by the AG system. For example, transformations are expressed as attribute grammars with a single, specially designated, attribute declaration for a copy of the tree being walked over. The only thing that remains to be specified is where the transformation differs from the original.

The AG notation allows us to avoid writing much boilerplate code, similar to other tree traversal approaches (Visser 2005, 2001; Lämmel and Peyton Jones 2003). The use of attributes sometimes also resembles reader, writer, and state monads. In practice, the real strength of the AG system lies in combining the separately defined tree traversals into one. For example, the EH type analysis repeatedly builds environments for kinds, types, datatypes, etc. Combined with the above idiomatic use this easily leads to many passes over the EH tree; something we'd rather not write by hand using monads (and monad transformers) or other mechanisms more suitable for single-pass tree traversals!

However, not all idiomatic use is supported by AG. For example, the need to pattern match on subtrees arises when case analysis on abstract syntax trees must be done. Currently this must be programmed by hand, and we would like to have automated support for it (as in Stratego (Visser 2005, 2001)).

The use of domain specific languages (DSL) We use various special purpose languages for subproblems: AG for tree traversals, Shuffle for incremental, aspect-wise, and better explainable development, Ruler for type systems. Although this means a steeper learning curve for those new to the implementation, in practice the used DSLs and their supporting tools effectively solve a identifiable design problem. There lies a balance here, but for us the used DSLs make development life easier.

Annotations We tend to extend languages with annotations. This is either to prevent keeping separate lookup tables (e.g. for arity of constructors), or to propagate information that is required in a later stage (e.g. type information of FFIs), or to track the origin of constructs (e.g. class declaration).

5.3 Related work

Clearly other Haskell compilers exist, most notably GHC (Marlow et al. 2004), which is hard if not impossible to match in its reliability and feature richness: UHC itself uses GHC as its main development tool.

Recently, JHC (Meacham 2009) and LHC (Himmelstrup et al. 2009) (derived from JHC) also take the full program analysis approach proposed by Boquist (Boquist and Johnsson 1996; Boquist 1999) as their starting point. LHC in its most recent incarnation is available as a backend to GHC, and in that sense is not a standalone Haskell compiler.

Already longer available alongside GHC are Hugs (Jones 2003) which was influential on Haskell as a language, NHC98 (Group 2007), and YHC (Shackell et al. 2009) derived from NHC98, all mature Haskell 98 compilers with extensions. Helium (Heeren et al. 2005) (also from Utrecht) does not implement full Haskell 98 but focuses on good error reporting, thereby being suitable for learning Haskell.

The distinguishing feature of UHC is its internal organization. UHC, in particular its internal aspect-wise organized framework, is designed to be (relatively) easy to use as a platform for research and education. In Utrecht students regularly use the UHC framework to experiment with. The use of AG and other tools also make UHC differ from other Haskell compilers, most of them written in Haskell or lower level languages.

5.4 Future work

We have recently released a first version of UHC. In the near future we intend to add support for better installation, in particular the use of Cabal, and add missing language features and libraries. On a longer time scale we continue working on full program analysis, the optimizations allowed by it, add classical analyses (e.g. strictness), improve the runtime system (own garbage collector). We welcome those who want to contribute in these or other areas of interest.

References

R. Bird and O. de Moor. *The algebra of programming*. Prentice Hall, 1996.

Richard S. Bird. Using Circular Programs to Eliminate Multiple Traversals of Data. *Acta Informatica*, 21:239–250, 1984.

H. Boehm. A garbage collector for C and C++. `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`, 2006.

H. Boehm and M. Weiser. Garbage Collection in an Uncooperative Environment. *Software Practice and Experience*, pages 807–820, Sep 1988.

Urban Boquist. *Code Optimisation Techniques for Lazy Functional Languages, PhD Thesis*. Chalmers University of Technology, 1999.

Urban Boquist and Thomas Johnsson. The GRIN Project: A Highly Optimising Back End For Lazy Functional Languages. In *Selected papers from the 8th International Workshop on Implementation of Functional Languages*, 1996.

Haskell' Committee. Haskell Prime. `http://hackage.haskell.org/trac/haskell-prime/`, 2009.

Atze Dijkstra. *Stepping through Haskell*. PhD thesis, Utrecht University, Department of Information and Computing Sciences, 2005.

Atze Dijkstra and S. Doaitse Swierstra. Ruler: Programming Type Rules. In *Functional and Logic Programming: 8th International Symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006*, number 3945 in LNCS, pages 30–46. Springer-Verlag, 2006.

Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. The Structure of the Essential Haskell Compiler, or Coping with Compiler Complexity. In *Implementation of Functional Languages*, 2007.

Karl-Filip Faxén. A Static Semantics for Haskell. *Journal of Functional Programming*, 12(4):295, 2002.

Jeroen Fokker and S. Doaitse Swierstra. Abstract interpretation of functional programs using an attribute grammar system. In Adrian Johnstone and Jurgen Vinju, editors, *Language Descriptions, Tools and Applications (LDTA08)*, 2008.

York Functional Programming Group. NHC98 Haskell Compiler. `http://haskell.org/nhc98/`, 2007.

Bastiaan Heeren, Arjan van IJzendoorn, and Jurriaan Hage. Helium, for learning Haskell. `http://www.cs.uu.nl/helium/`, 2005.

David Himmelstrup, Samuel Bronson, and Austin Seipp. LHC Haskell Compiler. `http://lhc.seize.it/`, 2009.

ISO. *Common language infrastructure (ISO/EIC standard 23271)*. ECMA, 2006.

Mark P. Jones. Typing Haskell in Haskell. In *Haskell Workshop*, 1999.

Mark P. Jones. Hugs 98. `http://www.haskell.org/hugs/`, 2003.

D.E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

D.E. Knuth. Literate Programming. *Journal of the ACM*, (42):97–111, 1984.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Types In Languages Design And Implementation*, pages 26–37, 2003.

Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

Simon Marlow, Simon Peyton Jones, and et. al. The Glasgow Haskell Compiler. `http://www.haskell.org/ghc/`, 2004.

John Meacham. Jhc Haskell Compiler. `http://repetae.net/computer/jhc/`, 2009.

Simon Peyton Jones. Compiling Haskell by program transformation: a report from the trenches. In *European Symposium On Programming*, pages 18–44, 1996.

Simon Peyton Jones. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell . In *Engineering theories of software construction, Marktoberdorf Summer School*, 2002.

Simon Peyton Jones. *Haskell 98, Language and Libraries, The Revised Report*. Cambridge Univ. Press, 2003.

Simon Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, pages 393–434, 2002.

Tom Shackell, Neil Mitchell, Andrew Wilkinson, et al. YHC York Haskell Compiler. `http://haskell.org/haskellwiki/Yhc`, 2009.

S. Doaitse Swierstra, P.R. Azero Alocer, and J. Saraiva. Designing and Implementing Combinator Languages. In *3rd Advanced Functional Programming*, number 1608 in LNCS, pages 150–206. Springer-Verlag, 1999.

GHC team. The New GHC/Hugs Runtime System. `http://citeseer.ist.psu.edu/marlow98new.html`, 1998.

Marcos Viera, S. Doaitse Swierstra, and Wouter S. Swierstra. Attribute grammars fly first class: How to do aspect oriented programming in haskell. In *International Conference on Functional programming (ICFP '09)*, New York, NY, USA, 2009. ACM Press.

Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, number 2051 in LNCS, pages 357–361. Springer-Verlag, 2001.

Eelco Visser. Stratego Home Page. `http://www.program-transformation.org/Stratego/WebHome`, 2005.