UHC/EHC structure

Jeroen Fokker, Atze Dijkstra

September 28, 2012

Contents

Haskell compilers are complex programs. Testimony to this observation is the Glasgow Haskell Compiler (GHC), which simultaneously incorporates many novel features, is used as a reliable workhorse for many a functional programmer, and offers a research platform for language designers. As a result, modifying GHC requires much knowledge of GHC's internals. In this experience report we describe the structure of the Essential Haskell Compiler (EHC) and how we manage complexity, despite its growth beyond the essentials towards a full Haskell compiler. Our approach partitions both language and its implementation into smaller, manageable steps, and uses compiler domain specific tools to generate parts of the compiler from higher level descriptions. As major parts of EHC have been written in Haskell both its implementation and use are reported about.

1 Introduction

Haskell is a perfect example of a programming language which offers many features improving programming efficiency by offering a sophisticated type system. As such it is an answer for the programmer looking for a programming language which does as much as possible of the programmer's job, while at the same time guaranteeing program properties like "well-typed programs don't crash". However, the consequence is that a programming language implementation is burdened by these responsibilities, and consequently becomes quite complex. Haskell thus also is a perfect example of a programming language for which compilers are complex. Testimony to this observation is the Glasgow Haskell Compiler (GHC), which simultaneously incorporates many novel features, is used as a reliable workhorse for many a functional programmer, and offers a research platform for language designers. As a result, modifying GHC requires much knowledge of GHC's internals.

In this writing we show how we deal with the complexity of compiling Haskell in the Essential Haskell (EH) Compiler (EHC). EH intends

- to compile full Haskell (the H in EH)
- to offer an implementation in terms of the essential, or desugared, core language constructs of Haskell (the E in EH)
- to provide a solid framework for research (i.e., extendable for experimentation) and education (another interpretation of the E in EH)

In particular the following areas require attention:

- **Implementation complexity** (section (page 2)) The amount of work a compiler has to do is a source of complexity. We organise the work as a series of smaller transformation steps between various internal representations.
- **Description complexity** (section (page 4)) The specification of parts of the implementation itself can become complex because low-level details are visible. We use domain specific languages which factor out such low-level details, so they are dealt with automatically.
- **Design complexity** (section (page 9)) Experiments with language features are usually done in isolation. We describe their implementation in isolation, as a sequence of language variants, building on top of each other.
- **Maintenance complexity** (section (page 10)) Actual compiler source, its documentation, and its specification tend to become inconsistent over time. We fight such inconsistencies by avoiding their main cause: duplication. Whenever two artefacts have to be consistent, we generate them from a common description.

In the next sections we explain how we deal with each of these complexities.

2 Coping with implementation complexity: transform

EHC is organised as a sequence of transformations between internal representations of the program being compiled. In order to keep the compiler understandable, we keep the transformations simple, and consequently, there are many. This approach is similar to the one taken in GHC. All our transformations are expressed as a full tree walk over the data structure, using a tool for easily defining tree walks (see section (page 4)). At each step in which the representation changes drastically we introduce a separate data structure (or "language"). figure (page 3) shows these languages and the transformations between them:

- **HS** (Haskell) is a representation of the program text as parsed. It is used for desugaring, name and dependency analysis, and making binding groups explicit.
- **EH** (Essential Haskell) is a simplified and desugared representation. It is used for type analysis and code expansion of class system related constructs.
- **Core** is a representation in an untyped *lambda*-calculus.
- **Grin** (Graph reduction intermediate notation) is a representation proposed by Boquist in which local definitions have been made sequential and the need for evaluation has been made explicit.
- **Silly** (Simple imperative little language) is a simple abstraction of an imperative language with an explicit stack and heap, and functions which can be called and tail-called.
- **C** is used here as a universal back-end, hiding the details of the underlying machine. Primitive functions are implemented here.

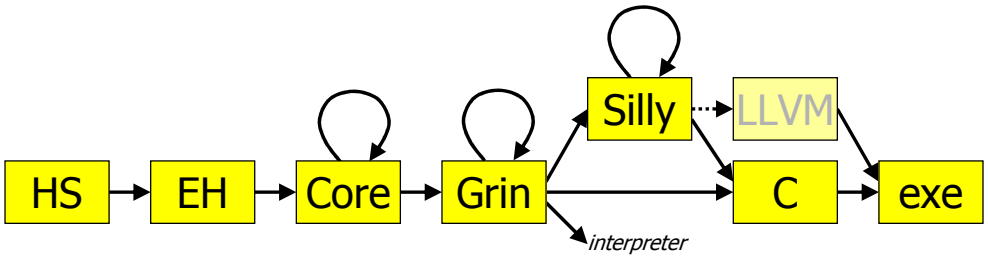

Figure 1: Intermediate languages and transformations in the EHC pipeline

- **LLVM** (Low level virtual machine) is an imperative language which, other than C, is *intended* to be a universal back-end. We have it under consideration as an alternative route to attain executable code.

As can be seen from the figure, the compilation pipeline branches after the Grin stage, offering different modes of compilation:

- Grin code can be interpreted directly by a simple (and thus slow) interpreter.
- Grin code can be translated to C directly. In this mode, the program is represented in a custom bytecode format, stored in arrays, and executed by an interpreter written in C. Its speed is comparable to that of Hugs.
- Grin code can be translated to executable code via transformations which perform global program analysis, and generate optimized Silly code, which can be further processed through either the C or LLVM route.

The transformations between the languages mentioned above bring the program stepwise to a lower level of representation, until it can be executed directly. Most of the simplification work however is done by the transformations that are indicated by a loop in figure (page 3), i.e., for which the source and target language are the same. We strive to have many small transformations rather than a few complicated ones. To give an idea, we list a short description of the more important of these transformations. Some of these are necessary simplifications, others are optimisations that can be left out.

- Transformations on the Core language include:
  - Cleanup transformations: *Eta-reduction*, *Eliminating trivial applications*, *Inline let alias*, *Remove unnecessary letrec mutual recursion*
  - *Constant propagation* and *Rename identifiers to unique names*
  - Lambda lifting, split up in: *Full laziness of subexpressions*, *Lambda/CAF globals passed as argument*, *Float lambda expressions to global level*
- Transformations on the Grin language include:
  - Transformations on separate modules: *Alias elimination*, *Unused name elimination*, *Eval elimination*, *Unboxing*, *Local inlining*
  - Transformations based on a global abstract interpretation that determines possible constructors of actual parameters: *Inline eval operation*, *Remove dead case alternatives and unused functions*, *Global inlining*
  - Transformations that remove higher-level constructs, such as splitting complete nodes into their constituent fields.

- Transformations on the Silly language include:
  - *Shortcut*: avoid unnecessary copying of local variables
  - *Embed*: map local variables to stack positions

3 Coping with description complexity: use tools

Haskell is well suited as an implementation language for compilers, among others because of the ease of manipulating tree structures. Still, if one needs to write many tree walks, especially if these involve multiple passes over complicated syntax trees, the necessary mutually recursive functions tend to become hard to understand, and contain large pieces of boilerplate code. In the implementation of EHC we therefore use a chain of preprocessing tools, depicted in figure (page 4).

We use the following preprocessing tools:

- **UUAGC** (Utrecht University Attribute Grammar Compiler), which enables us to specify abstract syntax trees and tree walks over them using an attribute grammar (AG) formalism.
- **Shuffle**, which deals with the compiler organisation and logistics of many different language features, and provides a form of literate programming.
- **Ruler**, a translator for an even more specialized language than AG, which enables a high-level specification of type inferencing, generating both AG code and *LaTeX* documentation.

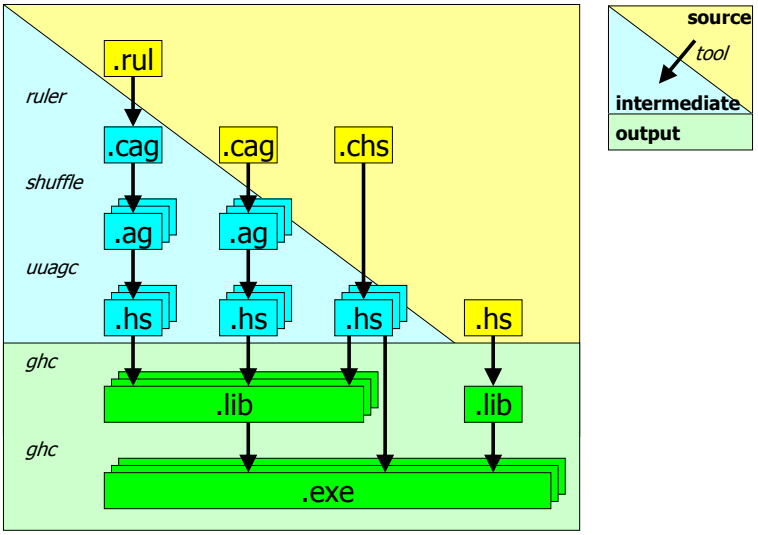

Figure 2: Chain of tools used to build EHC

In the remainder of this section we elaborate on the rationale of UUAGC (section (page 4)) and *Ruler* (section (page 8)). We illustrate their use with example code, which implements part of a Hindley-Milner type checker. In the section on UUAGC this is idealized toy code, but in the section on *Ruler* we show actual code taken from EHC for the same example. In section (page 9) and 5 we continue with the rationale and use of *Shuffle*.

3.1 UUAGC, a system for specifying tree walks

Higher-order functional languages are famous for their ability to parameterize functions not only with numbers and data structures, but also with functions and operators. The standard textbook

example involves the functions *sum* and *product*, which can be defined separately by tedious inductive definitions:

```
sum      []      = 0
sum      (x:xs)  = x + sum xs
product  []      = 1
product  (x:xs)  = x * product xs
```

This pattern can be generalized in a function *foldr* that takes as additional parameters the operator to apply in the inductive case and the base value:

```
foldr op e []      = e
foldr op e (x:xs)  = x `op` foldr op e xs
```

Once we have this generalized function, we can partially parameterize it to obtain simpler definitions for *sum* and *product*, and many other functions as well:

```
sum        =  foldr (+)  0
product    =  foldr (*)  1
concat     =  foldr (++) []
sort       =  foldr insert []
transpose  =  foldr (zipWith (:)) (repeat [])
```

The idea that underlies the definition of *foldr* (capturing the pattern of an inductive definition by adding 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. Functions that can be expressed in this way are called *catamorphisms* by Bird, and the collective extra parameters to *foldr*-like functions *algebras*. Thus, *((+),0)* is an algebra for lists, and *((++),[])* is another. In fact, every algebra defines a *semantics* of the data structure.

Outside circles of functional programmers and category theorists, an algebra is simply known as a "tree walk". In compiler construction, algebras could be very useful to define a semantics of a language, or bluntly said to define tree walks over the parse tree. This is not widely done, due to the following problems:

1. Unlike lists, which have a standard function *foldr*, in a compiler we deal with (many) custom data structures to describe the abstract syntax of a language, so we have to invest in writing a custom *fold* function first. 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 an alternative semantics: pretty-printed listings, warning messages, and various derived structures for internal use (symbol tables etc.). This can be done in one pass by having the semantic functions in the algebra return tuples, but this makes them hard to handle.

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 leaves of the parse tree to be able to evaluate them. This can be done by making the semantic functions in the algebra higher order functions, but this pushes the handling of algebras beyond human control.

5. Much of the work is just passing values up and down the tree. The essence of a semantics in the algebra is obscured by lots of boilerplate.

In short: the concepts of catamorphism and algebra apply here, but their encoding in Haskell is cumbersome and becomes prohibitively complex. Many compiler writers thus end up writing ad hoc recursive functions instead of defining the semantics by an algebra, or even resort to non-functional techniques. Others try to capture the pattern using monads. Some succeed in giving a concise definition of a semantics, often using proof rules of some kind, but loose executability. For the implementation they still need conventional techniques, and the issue arises whether the program soundly implements the specified semantics.

To save the nice idea of using an algebra for defining a semantics, we use a preprocessor for Haskell that overcomes the mentioned problems. It is not a separate language; we can still write auxiliary Haskell functions, and use all abstraction techniques and libraries. The preprocessor just allows a few additional constructs, which are translated into custom *fold*-like functions and algebras.

We describe the main features of the preprocessor here, and explain why they overcome the five problems mentioned above. For a start, the grammar of the abstract syntax of the language is defined in *DATA* declarations, which are like a Haskell *data* declaration with named fields, except that we do not have to write braces and commas and that constructor function names need not be unique. As an example, we show a fragment of EHC that represents a lambda calculus:

```
DATA Expr
  =  Var     name  :: Name
  |  Let     decl  :: Decl       body  :: Expr
  |  App     func  :: Expr       arg   :: Expr
  |  Lam     arg   :: Pat        body  :: Expr
DATA Decl
  =  Val     pat   :: Pat        expr :: Expr
DATA Pat
  =  Var     name  :: Name
  |  App     func  :: Expr       arg :: Expr
```

The preprocessor generates corresponding Haskell *data* declarations (making the constructors unique by prepending the type name, like `Expr_Var`), and more importantly, generates a custom *fold* function. This overcomes problem 1.

For any desired value we wish to compute from a tree, we can declare a "synthesized attribute" (the terminology goes back to Knuth). Attributes can be defined for one or more data types. For example, we can define that for all three datatypes we wish to synthesize a pretty-printed listing, and that expressions in addition synthesize a type and a variable substitution map:

```
ATTR Expr Decl Pat  SYN  listing   :: String
ATTR Expr           SYN  typ       :: Type
                         varmap    :: [(Name,Type)]
```

In the presence of multiple synthesized attributes, the preprocessor ensures that the semantic functions combine them in tuples, but in our program we can simply refer to the attributes by name. The attribute declarations of a single datatype can even be distributed over the program. This overcomes problem 2.

The value of each attribute needs to be defined for every constructor of every data type which has the attribute. These definitions of the semantics of the language are known as "semantic rules", and start with keyword *SEM*. An example is:

```
SEM Expr | Let
  lhs.listing =  "let " ++ @decl.listing ++ " in " ++ @body.listing
```

This states that the synthesized *listing* attribute of a *Let* expression can be constructed by combining the *listing* attributes of its *decl* and *body* children and some fixed strings. The @ symbol in this context should be read as "attribute", not to be confused with Haskell "as-patterns". The keyword *lhs* refers to the parent of the children *@decl* and *@body*, i.e., the nameless *Expr* at the **l**eft **h**and **s**ide of the grammar rule. At the left of the = symbol, the attribute to be defined is mentioned (here the

@ symbol may be omitted); at the right, any Haskell expression can be given. The example below shows the use of a *case* expression and an auxiliary function *substit*, applied to occurrences of child attributes. Also, it shows how to use the value of leaves (*@name* in the example), and how to group multiple semantic rules under a single *SEM* header:

```
SEM Expr
  | Var  lhs.listing  =  @name
  | Lam  lhs.typ      =  Type_Arrow (substit @body.varmap @arg.typ) @body.typ
  | App  lhs.typ      =  case @func.typ of
                           (Type_Arrow p b) -> substit @arg.varmap b
```

The preprocessor collects and orders all definitions into a single algebra, replacing the attribute references by suitable selections from the results of the recursive 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. As an example, it can serve to pass an environment (a lookup table that associates variables to types), which can be consulted when we need to determine the type of a variable:

```
ATTR Expr INH env :: [(Name,Type)]
SEM Expr
  | Var  lhs.typ  =  fromJust (lookup @name @lhs.env)
```

The value to use for the inherited attributes can be defined in semantic rules higher up the tree. In the example, *Let* expressions extend the environment which they inherited themselves with the new environment synthesized by the declaration, in order to define the environment to be used in the body:

```
SEM Expr
  | Let  body.env  =  @decl.newenv ++ @lhs.env
```

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

In practice, there are many situations where inherited attributes are passed unchanged as inherited attributes for the children. For example, the environment is passed down unchanged at *App* expressions. This can be quite tedious to do:

```
SEM Expr
  |  App  func.env  = @lhs.env
          arg.env   = @lhs.env
```

Since the code above is trivial, the preprocessor has a convention that, unless stated otherwise, attributes with the same name are automatically copied. So, the attribute *env* that an *App* expression inherited from its parent, is automatically copied to the children which also inherit an *env*, and the tedious rules above can be omitted. This captures a pattern that is often addressed by introducing a *Reader* monad. Similar automated copying is performed for synthesized attributes, so if they need to be passed unchanged up the tree, this does not need an explicit encoding, nor a *Writer* monad.

It is allowed to declare both an inherited and a synthesized attribute with the same name. In combination with the copying mechanisms, this enables us to silently thread a value through the entire tree, updating it when necessary. Such a pair of attributes can be declared as if it were a single "threaded" attribute. A useful application is to thread an integer value as a source for fresh variable names, incrementing it whenever a fresh name is needed during the tree walk. This captures a pattern for which otherwise a *State* monad would be needed.

The preprocessor automatically generates semantic rules in the standard situations described, and this overcomes problem 5.

3.2 Ruler, a system for specifying type rule implementations

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. Currently a formal description of Haskell, suitable for both the generation of an implementation and use in formal proofs, does not exist. For EH we make a step in that direction with *Ruler*, which allows us to have both an implementation and a type rule presentation with the guarantee that these are mutually consistent.

With *Ruler* we describe type rules in such a way that both a *LaTeX* rendering and an AG implementation can be generated from such a common type rule description. We demonstrate the use of *Ruler* by showing *Ruler* code for the Hindley-Milner type inferencing of function application *App* (see previous section for this and other names for expression terms). We omit a thorough explanation of the meaning of these fragments, as our purpose here is to demonstrate how we can describe these fragments with one common piece of *Ruler* source text. Also we do not intend to be complete in our description; we point out those parts corresponding to the distinguishing features of the *Ruler* system.

From a single source, to be discussed below, *Ruler* can both generate a *LaTeX* rendering for human use in technical writing:

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

Figure 3: Ruler example output

and its corresponding AG implementation, for further processing by UUAGC:

```
SEM Expr
  | App  (func.gUniq,loc.uniq1)
                        =  mkNewLevUID @lhs.gUniq
         func   .  knTy   =  [ mkTyVar @uniq1 ] `appArr` @lhs.knTy
         (loc.ty_a_,loc.ty_)
                        =  appUn1Arr @func.ty
         arg    .  knTy   =  @ty_a_
         loc    .  ty     =  @arg.tyVarMp |=> @ty_
```

The given rule describes the algorithmic typing of a function application in a standard lambda calculus with the Hindley-Milner type system. The rule involves four judgements: three premises and a conclusion. All judgements but the one involving the freshness of a type variable have the same structure as these all relate various properties of expressions: the conclusion about the function application *e1 e2*, the premises about the function *e1* and argument *e2*.

Ruler exploits this commonality by means of the *scheme* of a judgement, which can be thought of as the type of a judgement:

```
scheme expr =
    holes  [ node e: Expr, inh valGam: ValGam, inh knTy: Ty
           , thread tyVarMp: VarMp, syn ty: Ty ]
    judgeuse tex valGam ; tyVarMp.inh ; knTy :-.."e" e : ty ~> tyVarMp.syn
    judgespec valGam ; tyVarMp.inh ; knTy :- e : ty ~> tyVarMp.syn
```

The scheme declaration for expressions *expr* defines a common framework for the judgements of each *expr* term, such as *App* and *Lam* (lambda expression):

- *holes*: names, types and modifiers of placeholders (or *hole*s) for various properties, such as *e* and *valGam*
- *judgeuse tex* (unparsing): *LaTeX* pretty printing in terms of holes and other symbols, such as :- and ¿
- *judgespec* (parsing): concrete syntax for specifying a complete judgement.

Modifiers *node*, *inh*, *syn*, and *thread* are required when generating an AG implementation, to be able to turn a rule into an algorithm. The *thread* modifier introduces two holes with suffix *.inh* and *.syn*, corresponding to an AG threaded attribute. For a *LaTeX* rendering these modifiers are ignored, but additional formatting is required to map identifiers to *LaTeX* symbols, for example:

$$\begin{aligned} valGam &\mapsto \Gamma \\ ty &\mapsto \sigma \\ knTy &\mapsto \sigma^k \\ tyVarMp.\mathbf{inh} &\mapsto \mathcal{C}^k \end{aligned}$$

Figure 4: Ruler name mapping

We omit further discussion of lexical issues.

The rule for function application *App* now is defined by judgements introduced with the keyword *judge*:

```
rule e.app =
      judge tvarvFresh
      judge expr =  tyVarMp.inh ; tyVarMp ; (tvarv -> knTy)
                    :- eFun : (ty.a -> ty) ~> tyVarMp.fun
      judge expr =  tyVarMp.fun ; valGam ; ty.a
                    :- eArg : ty.a ~> tyVarMp.arg
      -
      judge expr =  tyVarMp.inh ; valGam ; knTy
                    :- (eFun  eArg) : (tyVarMp.arg ty) ~> tyVarMp.arg
```

For each judgement its scheme is specified (*expr* in the example). The *judgespec* of the corresponding scheme is used to check the concrete syntax and to bind the holes of the judgement to the concrete values specified by the judgement. From this rule definition a *LaTeX* rendering can straightforwardly be generated.

For the generation of an AG implementation we need information as specified by hole modifiers. In an AG implementation the structure of the tree drives the choice of which rule to apply. One of the holes needs to correspond to a node of such a tree; the modifier *node* specifies which. Other holes correspond to attributes, which have a direction: top-down (inherited, indicated by modifier *inh*) bottom-up (synthesized, indicated by *syn*) or both (indicated by *thread*).

The judgement with scheme *tvarvFresh* is an example of a judgement which does not fit into a tree structure as required by AG: it does not refer to a *node* hole. For such schemes, called *relation*s, an explicit AG implementation must be provided. We omit further discussion of relations.

Finally, *Ruler* also provides support for incremental language specification, which we discuss in section (page 9).

4 Coping with design complexity: grow stepwise

To cope with the many features of Haskell, EHC is constructed as a sequence of compilers, each of which adds new features. This enables us to experiment with non-standard features. figure (page 10)

shows the standard and experimental features currently introduced in each language variant. The sequence is a didactical choice of increasingly complex features; it is not the development history. Every compiler in the sequence can actually be built out of the repository.

Each language variant in the sequence is described as a delta with respect to the previous language. Usually this delta is a pure addition, but other combinations are possible when:

- language features interact
- the overall implementation and individual increments interact: an increment is described in the context of the implementation of preceding variants, whereas such a context must anticipate later changes.

Conventional compiler building tools are neither aware of partitioning into increments nor aware of their interaction. We use a separate tool, called *Shuffle*, to take care of such issues. We describe *Shuffle* in the next section.

	Haskell	**extensions**
1	*lambda* -calculus, type checking	
2	type inferencing	
3	polymorphism	
4		higher ranked types, existentials
5	data types	
6	kind inferencing	kind signatures
7	records	tuples as records
8	code generation	GRIN
9	class system	
10		extensible records
11	type synonyms	
12		explicit passing of implicit parameters *
13		higher order predicates *
14–19		*reserved for other extensions* *
20	module system	
95	class instance deriving *	
96		exception handling
97	numbers: Integer, Float, Double	
98	IO	
99	the rest for full Haskell *	

For each language variant in the sequence, various artefacts are created, such as example programs, a definition of the semantics, an implementation, and documentation. figure (page 14) shows some of these artefacts for some language variants. The first row shows an example program for each language variant. The second row shows a description of part of the semantics of the language variants (the type rule for functional application), by way of the *LaTeX* rendering of the type rule generated by *Ruler*. The third row shows the implementation of this type rule in the compiler, by way of the AG output generated by *Ruler* (from the same source). Example language variants shown in the columns of figure (page 14) are EH1 (simply explicitly typed *lambda*-calculus), EH3 (adding polymorphic type inference), and EH4 (adding higher-ranked types).

5 Coping with maintenance complexity: generate, generate and generate

For any large programming project the greatest challenge is not to make the first version, but to be able to make subsequent versions. In order to facilitate change, the object of change should be

isolated and encapsulated. Although many programming languages support encapsulation, this is not sufficient for the construction of a compiler, because each language feature influences not only various parts of the compiler (parser, structure of abstract syntax tree, type system, code generation, runtime system) but also other artefacts such as specification, documentation, and test suites. Encapsulation of a language feature in a compiler therefore is difficult, if not impossible, to achieve.

We mitigate the above problems by using *Shuffle*, a separate preprocessor. In all source files, we annotate to which language variants the text is relevant. *Shuffle* preprocesses all source files by selecting and reordering those fragments (called *chunk*s) that are needed for a particular language variant. Source code for a particular Haskell module is stored in a single "chunked Haskell" (.chs) file, from which *Shuffle* can generate the Haskell (.hs) file for any desired variant (see figure (page 4), where the stacks of intermediate files denote various variants of a module). Source files can be chunked Haskell code, chunked AG code, but also chunked *LaTeX* text and code in other languages we use.

Shuffle behaves similar to literate programming tools in that it generates program source code. The key difference is that with the literate programming style program source code is generated out of a file containing program text plus documentation, whereas *Shuffle* combines chunks for different variants from different files into either program source code or documentation.

Shuffle offers a different functionality than version management tools: these offer historical versions, whereas *Shuffle* offers the simultaneous handling of different variants from a single source.

For example, for language variant *2* and *3* (on top of *2*) a different wrapper function *mkTyVar* for the construction of the internal representation of a type variable is required. In variant *2*, *mkTyVar* is equal to the constructor `Ty_Var`:

```
mkTyVar :: TyVarId -> Ty
mkTyVar tv = Ty_Var tv
```

However, version *3* introduces polymorphism as a language variant, which requires additional information for a type variable, which defaults to `TyVarCateg_Plain` (we do not further explain this):

```
mkTyVar :: TyVarId -> Ty
mkTyVar tv = Ty_Var tv TyVarCateg_Plain
```

These two Haskell fragments are generated from the following *Shuffle* source:

```
%%[2.mkTyVar
mkTyVar :: TyVarId -> Ty
mkTyVar tv = Ty_Var tv
%%]

%%[3.mkTyVar -2.mkTyVar
mkTyVar :: TyVarId -> Ty
mkTyVar tv = Ty_Var tv TyVarCateg_Plain
%%]
```

The notation `%%[2.mkTyVar` begins a chunk for variant 2 with name *mkTyVar*, ended by `%%]`. The chunk for `3.mkTyVar` explicitly specifies to override `2.mkTyVar` for variant 3. Although the type signature can be factored out, we refrain from doing so for small definitions.

In summary, *Shuffle*:

- uses notation `%%[` ... `%%]` to delimit and name text chunks
- names chunks by a variant number and (optional) additional naming
- allows overriding of chunks based on their name
- combines chunks upto an externally specified variant, using an also externally specified variant ordering.

6 Related work

Compiler building environments Various compiler construction environments exist, originally developed some time ago. We mention Cocktail, Eli and Gentle. Of these environments Cocktail provides the most comprehensive and best maintained set of tools for lexical analysis, parsing, attribute grammars and tree transformations. With the exception of Cocktail these environments are focussed upon development in C; Cocktail allows codegeneration for various imperative languages. More specifically for Java, Polyglot is an extensible compiler frontend, with many experimental Java extensions.

In contrast, our toolset is built upon Haskell, and wherever possible exploits Haskell's assets such as its strong type system to provide various combinator libraries for (e.g.) parsing instead of traditional generator based solutions. Haskell is also used to specify attribute computations. This allows our tools to be relative lightweight yet comprehensive. It also strongly ties our tools to Haskell, which is benificial in our view because of the compact and descriptive nature of Haskell code fragments.

Tree based systems and Attribute Grammar systems Computations over abstract syntax trees and tree transformation form a major part of any compiler; Cocktail also incorporates an attribute grammar system, based on work by Kastens, as well as tools for transforming trees. Stratego (based on ASF/SDF) is a specialized tool for tree transformations. JastAdd is an attribute grammar and tree rewrite system built on top of Java, used to build a Java compiler.

In contrast, our attribute grammar system is also used for the many tree transformations present in EHC; support for making a modified replica of the tree being analysed makes this work suprisingly well. Occasionally we miss pattern matching features offered by specialized tree transformation tools, but on the other hand we are not limited by the sometimes restricted computational expressiveness offered by these tools.

Declarative specification Ruler intends to bridge the gap between formal specifications and their implementation, like Tinkertype. Ott provides a typerule based frontend, not for an implementation but for various theorem proving systems. Twelf is a theorem proving system, amongst others used to ultimately specify ML formally in order to proof type safety. Ruler, on the other hand, is currently only used as an implementation tool and documentation tool.

Embedded solutions All tools mentioned sofar are external tools, in the sense that they generate from a separate specification for a specific implementation language. Monads often have been used for Haskell embedded attribution, e.g. a reader monad corresponds directly to inherited attribution. A similar effect can be accomplished with boilerplate code avoiding approaches. Both approaches work fine for relatively small examples but do not scale well when separate computations have to be merged into one, for example when such computations need each others (intermediate) results.

7 Experiences

Development and debugging The partitioning into variants is helpful for both development and debugging. It is always clear to which variant code contributes, and if a problem arises one can use a previous variant in order to isolate the problem. Experimentation also benefits because one can pick a suitable variant to build upon, without being hindered by subsequent variants.

However, on the downside, there are builtin systemwide assumptions, for example about how type checking is done. We are currently investigating this issue in the context of *Ruler*.

Use in research and education EHC is constructed as a library and a toplevel compiler driver (see figure (page 4)), facilitating the use of the implementation of EHC by other programs.

We intend to use the first three language variants (figure (page 10)) in our basic course on compiler construction, thus providing students with a realistic integrated introduction to language design, compiler implementation, and software engineering. This approach is similar to that in Pierce's textbook, however, in contrast we focus on a realistic implementation of full Haskell instead of small independent implementations of isolated type systems.

Improvements Although our approach to cope with complexity indeed leads to the advocated benefits, there is room for improvement:

- **Ruler and type rules** With *Ruler* we generate both AG and LaTeX. *Ruler* notation, AG, and *LaTeX* have a similar structure. Consequently *Ruler* does not hide as much of the implementation as we would like. We are investigating a more declarative notation for *Ruler*.
- **Loss of information while transforming** With a transformational approach to different intermediate representations, the relation of later stages to earlier available information becomes unclear. For example, by desugaring to a simpler representation, source code of the user program is reordered and the original source location has to be propagated as part of the AST. Such information flow patterns are not yet automated.
- **High level description and efficiency** Using a high level description usually also provides opportunities to optimise at a low level. For attribute grammars a large body of optimisations are available, some of which are finding their way into our AG system.
- **Stepwise approach versus aspectwise approach** EH's stepwise approach imposes a fixed order in which language constructs are implemented on top of each other. Ideally one should be able to arbitrarily combine separate language constructs as aspects (independent implementation fragments), but interaction between language constructs hinders this flexibility. We are investigating the use of aspects in the context of *Ruler*.

Status and plans We are working towards a release of EHC as a Haskell compiler: variant 99 in the sequence. At the moment, we can compile a prelude and run programs with a bytecode interpreter. We intend to work on AG optimisations, on using LLVM as a backend, and on GRIN global transformations.

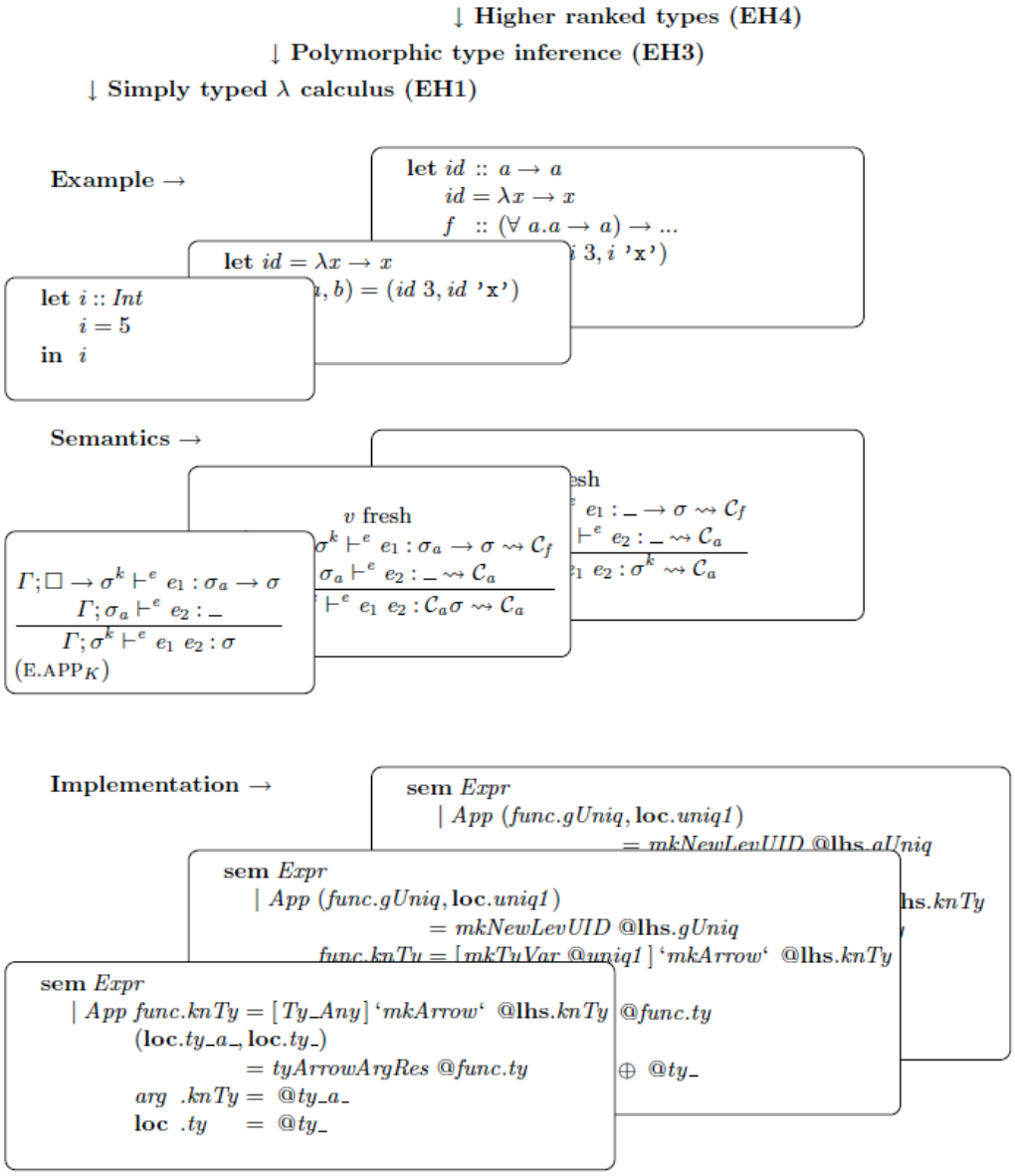

Figure 5: Examples of created artefacts (rows) for various language variants (columns)