Experience Report: The Structure of the Essential Haskell Compiler, or Coping with Compiler Complexity

author info removed for reviewing

–

–

Abstract

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 a steep learning curve for getting acquainted with 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 becomes complex as a result.

In this experience report we show how we deal with this compiler complexity in the context of the Essential Haskell (EH) Compiler (EHC) (Dijkstra 2004, 2005), and in particular the following issues:

- **Language features.** (Section 2) Language features usually are experimented with in isolation. We describe their implementation in isolation, as a sequence of language variants, building on top of each other.
- **Maintenance.** (Section 3) Actual compiler source, its documentation and specification tend to become inconsistent over time. We deal with inconsistencies by avoiding its main cause: duplication. Whenever two artefacts have to be consistent, we generate these from a common description.
- **Description complexity.** (Section 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 those low-level details which can be dealt with automatically.
- **Implementation complexity.** (Section 5) The amount of work a compiler has to do is a source of complexity as well. We use a dataflow through many relatively small transformations, between various internal representations. Although not novel (Peyton Jones et al. 1992; Peyton Jones and Santos 1994; Peyton Jones 1996), we mention it for completeness.

We focus on the overall organisation and discuss the benefits and drawbacks of it, leaving out references to well known tools and concepts.

2. Coping with design complexity: stepwise grow a language

To cope with the many features of Haskell, EHC is designed as a sequence of compilers, each of which adds new features. This enables us to experiment with non-standard features. Figure 1 shows all 11 language variants we currently recognize, with the standard and experimental features they introduce. The subdivision reflects a didactical choice of increasingly complex features; it is not the development history of the project. Every compiler in the series can actually be built out of the repository.

For each compiler in the series, various artefacts are created: a definition of the semantics, an implementation, example programs, documentation, etcetera. Figure 2 shows some instances. The language in the first column introduces the starting point for subsequent languages. It features the simply typed λ-calculus, where all defined values require an accompanying type signature, demonstrated by the example. For the description of the corresponding compiler we require the static semantics of all language constructs, their implementation and documentation. The semantics is defined in terms of type rules and the implementation in terms of attribute grammars.

The next two columns in Figure 2 show the incorporation of polymorphic type inference and higher ranked types, respectively. Actually, part of the latter related to the propagation of quantified types is experimented with as separate variant (Dijkstra and Swierstra 2006a), but is shown here to demonstrate the increase in complexity of both the type rules and the corresponding implementation.

The tool architecture was designed in such a way that the description for each language variant n consists of the delta with respect to language $(n-1)$. Usually this delta is a pure addition, but there is interaction between subsequent variants when:

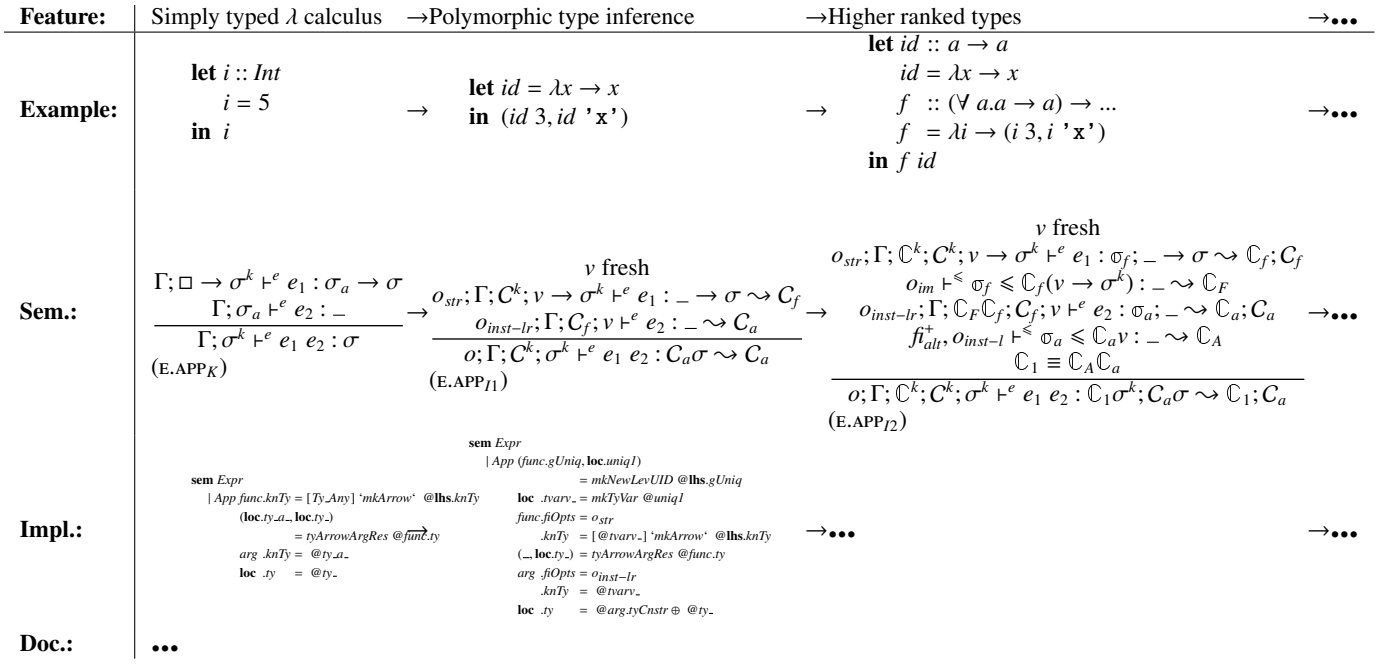

Figure 2. Languages design steps

	Haskell	extensions
1-4	λ-calculus, polymorphism, type checking/inferencing	higher ranked types, existentials
5	data types	
6	kind inferencing	kind signatures
7	records	tuples as records
8	code generation	GRIN
9	classes	explicit implicit parameters
10		extensible records
11 ...	type synonyms, modules Integer, Float, deriving, IO, ...	
...		

Figure 1. EH all language variants

- 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 their interaction. We use a separate tool, called *Shuffle*, to take care of such issues. We describe *Shuffle* in the next section.

3. 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. One must be prepared for change, and in order to facilitate change, the object of change should be isolated and encapsulated. Although many programming languages support encapsulation, this is not enough for the construction of a compiler, because one language feature influences not only different parts of a 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 files can be (chunked) Haskell code, (chunked) LaTeX text, but also code in other languages we use (see Figure 3).

Shuffle behaves similar to literate programming (Knuth 1984) 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: version management offers historical versions, whereas *Shuffle* offers the simultaneous handling of different variants from one source.

For example, for language variant 2 and 3 (on top of 2) a different Haskell wrapper function *mkTyVar* for the construction of the compiler internal representation of a type variable is required. In variant 2, *mkTyVar* is equivalent 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
```

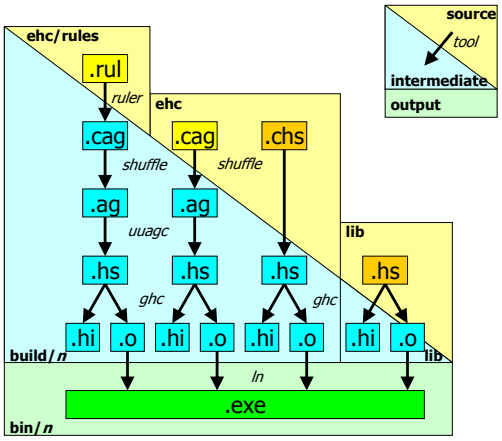

Figure 3. EHC toolchain

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

```
%%[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.

4. Coping with formalisation complexity: domain specific languages

Shuffle deals with the compiler organisation and logistics of many different language features. For the description and implementation of individual features we use the following domain specific languages:

- **Attribute Grammar (AG).** For specifying abstract syntax trees (AST) and tree walks over ASTs (Swierstra et al. 1999; Dijkstra and Swierstra 2004; Baars 2004; Dijkstra 2005).
- **Ruler.** For specifying type rules (Dijkstra and Swierstra 2006b).

Both *AG* and *Ruler* have their place in Figure 3. *Ruler* also is aware of language variants in order to be able to render type rules taking into account differences between language variants; hence it precedes *Shuffle* in the tool pipeline.

Both *AG* and *Ruler* earn their place in our toolchain because we can't do without. With *AG* we specify –in essence– folds over data types, without the need to specify boilerplate plumbing as well. With *Ruler* we specify type rules in such a way that both *AG* and LaTeX can be generated; we do not get lost in inconsistencies between semantics specification and compiler implementation, and LaTeX obscurities.

Both *AG* and *Ruler* also support growing definitions, or aspects. We demonstrate this for *AG*, by showing the additional AG fragments for the code generation aspect of variant 8 in Figure 1:

data *Expr*
  | *App func* : *Expr*
  arg : *Expr*

attr *Expr* [|| *cexpr* : *CExpr*]

sem *Expr*
  | *App* **lhs**.*cexpr* = *CExpr_App* @*func.cexpr* @*arg.cexpr*

From EH abstract syntax *Expr* (see Figure 4) we generate Core abstract syntax *CExpr* for function application. We omit further explanation of this fragment, but point out that this fragment independently of other *AG* fragments specifies how to compute the code generation aspect *cexpr* for function applications. The *AG* compiler then combines such separate specifications and builds one fold function, to be used on EH abstract syntax.

5. Coping with internal complexity: transformations

The many jobs a compiler has to do usually can be organised into a pipeline of internal representations and transformations. Figure 4 shows EHC's pipeline, with its major intermediate representations:

- **Haskell** A representation of the concrete program text, used for desugaring, name and dependency analysis, and the regrouping of definitions accordingly.
- **Essential Haskell** A simplified and desugared representation, used for type analysis, code generation, and code expansion of class system related constructs.
- **Core** An untyped λ-calculus representation, used for lambda-lifting.
- **GRIN** A GRIN representation (Boquist 1999), representing an almost imperative program, used for global program analysis.
- **Silly** A simple imperative language abstraction, used to generate C or any other imperative language such as input for LLVM (Lattner 2007).

Most of the intermediate representations can be pretty printed and parsed again, thus facilitating both debugging and experimenting. In Figure 4 the vertical arrows indicate this.

6. 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 go a variant back in order to isolate the problem. Experimentation

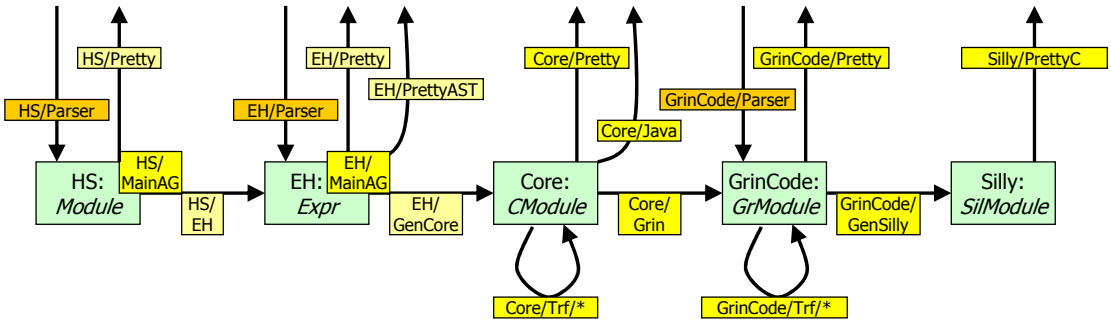

Figure 4. EHC toolchain

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 system wide assumptions, for example about how type checking is done. We are currently investigating this issue in the context of *Ruler*.

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 (Essential) Haskell representation, sourcecode gets reshuffled and their original source location has to be propagated correctly 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, of which some are finding their way into our AG system.

Status and plans We are working towards a release as a Haskell compiler: the last variant of the whole sequence. Compilation of a Prelude succeeds, and we run programs with a GRIN based bytecode interpreter. We intend to work on AG optimisations, use LLVM (Lattner 2007) as a backend, and GRIN global transformations.

References

Arthur Baars. Attribute Grammar System. `http://www.cs.uu.nl/groups/ST/Center/AttributeGrammarSystem`, 2004.

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

Atze Dijkstra. EHC Web. `http://www.cs.uu.nl/groups/ST/Ehc/WebHome`, 2004.

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

Atze Dijkstra and S. Doaitse Swierstra. Typing Haskell with an Attribute Grammar. In *Advanced Functional Programming Summerschool*, number 3622 in LNCS. Springer-Verlag, 2004.

Atze Dijkstra and S. Doaitse Swierstra. Exploiting Type Annotations. Technical Report UU-CS-2006-051, Department of Computer Science, Utrecht University, 2006a.

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, 2006b.

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

Chris Lattner. The LLVM Compiler Infrastructure Project. `http://llvm.org/`, 2007.

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 and Andre Santos. Compilation by Transformation in the Glasgow Haskell Compiler. `http://citeseer.ist.psu.edu/peytonjones94compilation.html`, 1994.

Simon Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Phil Wadler. The Glasgow Haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, 1992.

S. Doaitse Swierstra, P.R. Azero Alocer, and J. Saraiava. Designing and Implementing Combinator Languages. In Doaitse Swierstra, Pedro Henriques, and José Oliveira, editors, *Advanced Functional Programming, Third International School, AFP'98*, number 1608 in LNCS, pages 150–206. Springer-Verlag, 1999.