EHC Architecture

Jeroen Fokker

September 6th, 2006

This report describes the architecture of EHC, the Extensible Haskell Compiler as developed by the Software Technology group of Utrecht University.

In section 1 we introduce the tools which are used to build the compiler: ghc, shuffle, uuagc and ruler.

In section 2 we describe the main data flow of EHC, as a program is transformed from Haskell to executable code.

In section 3 we enumerate the source files and their dependencies.

In section 4 we describe the layout of the file system organization.

1 Tool chains

EHC is a Haskell compiler which itself is written in Haskell. Although in the future we might be able to bootstrap the compiler to compile itself, for the moment we use *ghc* to compile the compiler.

Only a few modules of the EHC source are actually written directly in Haskell. Most of the source is preprocessed by one or more tools. Figure 1 shows the way in which the tools generate intermediate files, which ultimately are linked together to build the EHC executable.

A short description of the tools follows.

- **ghc**
  The Glasgow Haskell compiler takes a Haskell module (file extension *hs*) and generates standard binary object code (file extension *o*). Many object code modules can be linked together with **ln**, but this is normally done automatically by ghc. Also, ghc generates files containing information for other modules that want to import the module (file extension *hi*).

- **shuffle**
  There are many views of EHC, each of which adds new features to its predecessor. Instead of maintaining the source code of each view separately, code for all views of a module is put together in a single file, known as a 'chunked Haskell script' (file extension *chs*). A chunked file contains meta information as to which chunks are needed for which view.

  A tool named 'shuffle' extracts the chunks that belong to any desired view and generates a normal Haskell script containing only those chunks.

  When invoked with different options, shuffle is also able to generate documentation from the source files, thus enable a 'literate programming' style of documentation.

- **uuagc**
  Recursive treewalks over an abstract data structure can be elegantly modeled using attribute grammars. Information passes down the tree as *inherited* attributes, and one or more semantic values can be derived as *synthesized* attributes.

  The 'Utrecht University Attribute Grammar Compiler' (uuagc) takes a description of an abstract syntax tree and its semantics, and generates Haskell functions that perform the necessary treewalks. Source code for uuagc is stored in files with extension *ag*.

  In the same way as for Haskell modules, different views of an AG compilation unit can be stored in a single, 'chunked' file. The shuffle tool is used to extract the desired chunks from a chunked attribute grammar files (file extension *cag*) to a file that subsequently is processed by uuagc.

- **ruler**
  The derivation of the type of an expression can be described as a treewalk over the expression, and therefore the AG-language can be used to describe typing algorithms. However, in the literature typing rules are often described in an even more high level language, involving judgements separated by horizontal rules.

  A tool named 'ruler' takes a description in this style, and generates the corresponding attribute grammar. As the ruler language contains a built-in mechansim for different views, no need for 'chunked' ruler programs is needed.

  When invoked with different options, ruler is also able to generate typeset descriptions of the rules, which are not only beautifully laid out, but also guaranteed to be consistent with the generated compiler modules.

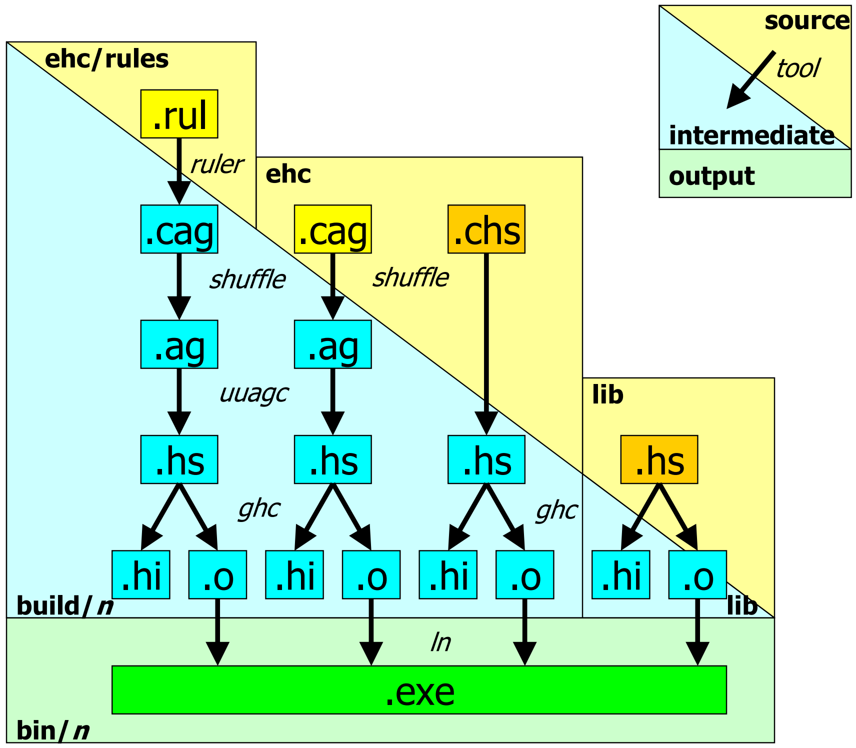

Figure 1: Tools in the EHC project

Figure 1 also shows the directories in which various files can be found:

- **Source files** (shown shaded yellow) are kept in a separate directory for each product:
  - chunked Haskell and chunked AG sources for the EHC frontend are in directory ehc
  - chunked Haskell and chunked AG sources for the EHC backend are in directory grinc
  - Ruler sources for EHC are in subdirectory ehc/rules
  - chunked Haskell and chunked AG sources for the Grin interpreter are in directory grini
  - library (non-chunked) Haskell modules written for, but not specific to this project are in directory libutil

- **Intermediate files** (shown shaded blue) are generated by the described tools, in a separate directory for each view as extracted by shuffle:
  - all intermediate files (un-chunked AG and HS files generated by shuffle and ruler, HS files generated by uuagc, HI and O files as generated by ghc) for view n are stored in directory build/n (for n between 1 and 12)
  - compiled views of the library modules are also stored in directory libutil

- **Output files** (shown shaded green)
  - all output files (executable for EHC compiler, Grin interpreter and compiler) for view n are stored in directory bin/n

  A detailed description of the file system layout is given in section 4.

2 EHC Main dataflow

Figure 2 shows the main data flow of EHC. The compilation process consists of various stages in which the program is transformed. There are multiple entry- and exit-points:

- Input can either be a Haskell program, a program in the custom EH notation, or a more low-level program in the Grin language. Input type is recognized by file extension (`hs`, `eh` or `grin`).
- Output can either be pretty printed Haskell, pretty printed EH with derived type annotations, pretty printed Core, pretty printed Grin, or ready-to-compile C code. Output type is selected by the `-c` compiler option.

The picture shows various intermediate data types in green boxes, and transformations between them.

- **HS:***Module* represents a Haskell module. There is a parser for it, and a single transformation which in one traversal performs both pretty printing and conversion to EH.
- **EH:***Expr* represents an expression in the EH custom language. In fact, this can be a program, as a program is just the expression from the body of `main` with many local definitions. There is a parser for it, and a single transformation which in one traversal does the typing, pretty printing, and conversion to Core.
- **Core:***CModule* represents a module in the Core language. Many conversions are available as a Core-to-Core transformation (e.g. lambda lifting, constant propagation). There are separate conversions to pretty printed output, input to a simulator written in Java, and a Grin datastructure.
- **GrinCode:***GrinModule* represents a module in Grin, a Graph Reduction Intermediate Notation. Many conversions are available as a Grin-to-Grin transformation (e.g. inlining of eval, pruning sparse case distinctions). There are separate conversions to pretty printed output, and a Silly datastructure.
- **Silly:***SilModule* represents a module in a generic imperative language, that can be converted to C (and in the future possibly to other imperative languages).

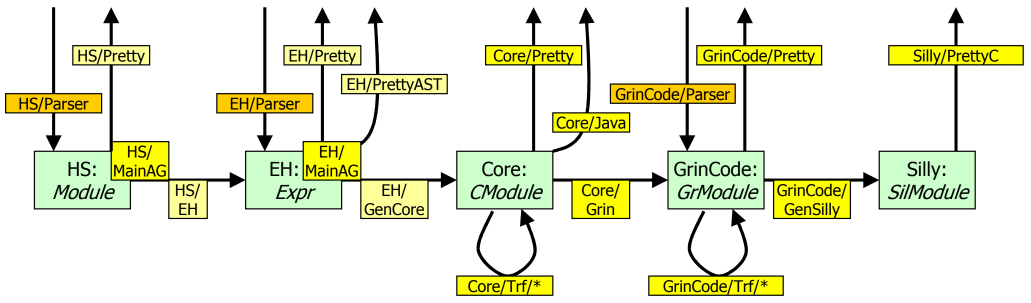

Figure 2: EHC main data flow

3 Source file dependencies

Figure 3 shows all source files of EHC. The files are organized in columns based on whether they describe syntax, describe semantics, or contain auxiliary functions. The files are organized in rows based on the subsystem they belong to. Arrows indicate inclusion as done by the AG-compiler.

The ordering in **columns** is as follows:

- Column *syntax, AG-unit* shows AG-units that describe the data structures for various tree structures. These units are compiled by the AG-compiler using the `-dr` option. The output of the AG-compiler is a Haskell module containing only data declarations. The files contain merely headers, and for the actual data structure import a corresponding file from the next column.

- Column *syntax, import* shows files that contain the actual data structure definitions. These files are not standalone AG-units, and thus no corresponding Haskell files are generated. They are imported by the syntax AG-units in the first column, but also by the semantics AG-units in the third column.

  The data types that are defined in these files are enumerated in the green box above the file name. List types are conventionally named by appending ‘s’ or ’L’ to the base type name.

- Column *semantics, AG-unit* shows AG-units that contain attribute definitions for the various tree structures. These units are compiled by the AG-compiler using the `-cfspr` option. The output of the AG-compiler is a Haskell module containing catamorphisms for the data structures, semantic functions and their signatures, and (in comment) a pretty printed list of all attributes.

- Column *semantics, import* shows files that are imported by the semantics AG-units. The files themselves are not standalone AG-units, and thus no corresponding Haskell files are generated. Import is handled by the AG-compiler as textual inclusion. This mechanism is used for two purposes:
  - (e.g., in the ‘HS’ and ‘EH’ subsystems:) to split up a large file in multiple files, each describing a few attributes
  - (e.g., in the ‘Type’ and ‘Core’ subsystems:) to use an attribute definition in more than one AG-unit. Note that in this case, the code for the attribute definitions is duplicated in the semantic function for each AG-unit in which is is imported.

- Column *auxiliary* shows files containing auxiliary Haskell functions. These functions will normally be:
  - a `main` function, and any function that is called from there
  - functions that are used in the attribute definitions

  As the AG-compiler does not interpret Haskell code in attribute definitions, files in this column are ignored by the AG-compiler.

The ordering in **rows** is according to subsystem. Some of these (shown in bold face in the list below) are a chain in the main data flow, the others are auxiliary.

- Row *Base* shows a few auxiliary files common to the whole system.
- Row *Scanner* shows files that perform the tokenization of input into symbols, that can be parsed by other subsystems.
- Row *Gam* shows an auxiliary file for manipulating environments (commonly denoted as Γ).
- Row **HS** shows files for reading and storing Haskell programs, and transforming them to the custom EH language.
- Row **EH** shows files for reading and storing programs in the EH custom language, checking and/or infering their types, and transforming them to the EH Core language.
- Row *Type* shows files for storing and manipulating types.
- Row **Core** shows files for storing programs in the EH Core language, doing transformations on them, and transforming them to a more imperative language Grin.
- Row *Error* shows files for manipulating error messages. An abstract syntax for (lists of) error messages is defined, and a semantic function to pretty print it.
- Row *libutil* shows some auxiliary Haskell files that are written for this project, but not specific to EHC. These files are ordinary (non-chunked) Haskell files.
- Row *grini* shows files for a stand-alone Grin interpreter, that can read and execute Grin programs.
- Row **GrinCode** shows files for reading and storing programs in Grin, doing transformations on them as described by Boquist, and transforming them to an imperative language.
- Row **Silly** shows files for storing and manipulate programs in a Simple Imperative little Language, which can be easily converted to C and other imperative languages.
- Row *CmmCode* shows an alternative back-end to Grin, generating C−− programs (this subsystems is not maintained anymore).

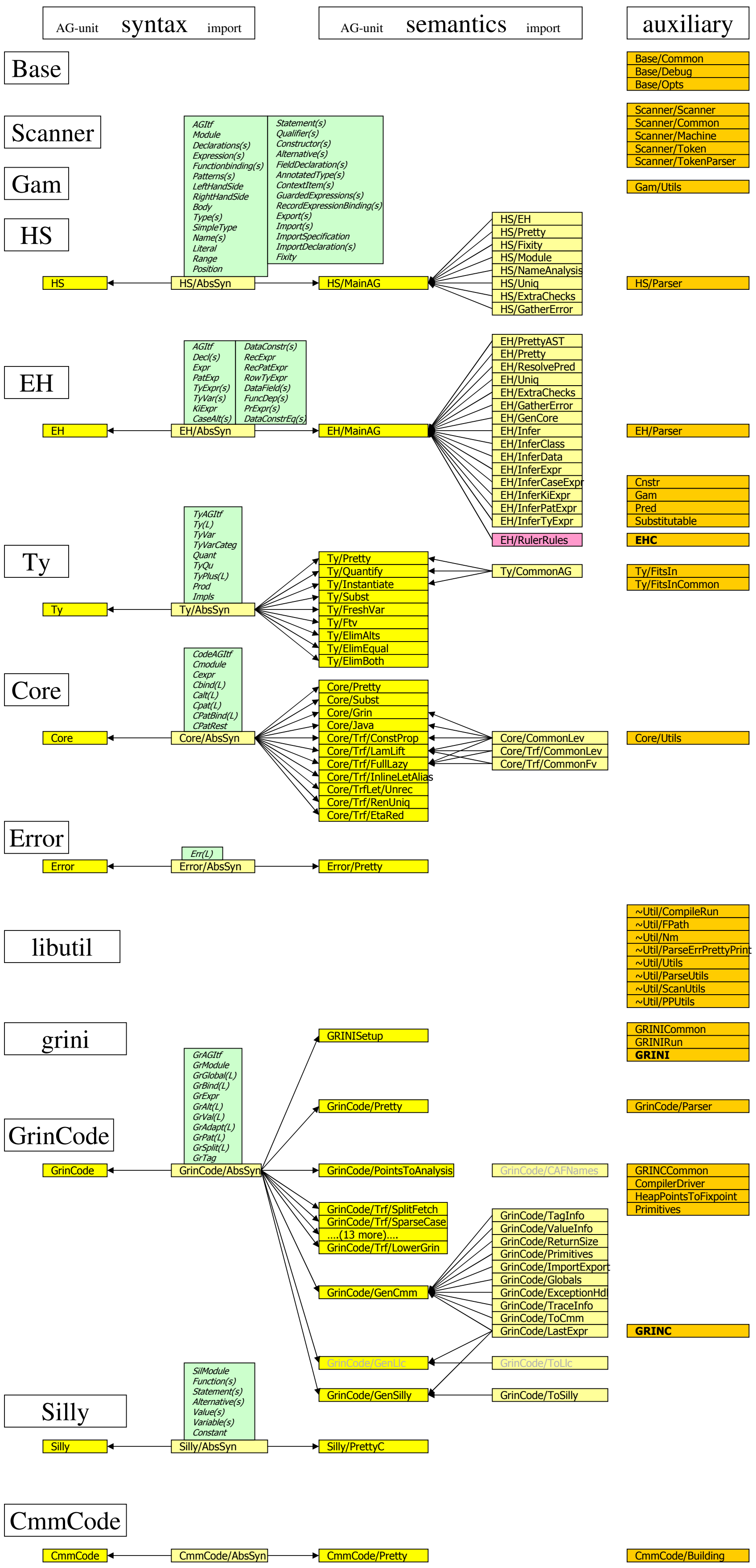

Figure 3: EHC source files, and their inclusion by the AG-compiler

4 File system layout

Figure 4 shows the organization of the EHC files in directories. Files shown in *yellow* are the original source files that you get when you download the system. Files shown in *blue* are intermediate files that are generated when you type `make`. Files shown in *green* are the final products that are generated.

The source directory **src** is structured as follows. The twelve subsystems described in section 3 are stored in three subdirectories *ehc*, *grinc* and *grini*. This is a historical remain from the time that the frontend and the backend were separate products. Hence, these directories are summarized by **P** (for 'product') in the picture. The *ehc* and *grinc* product directories have subdirectories for the twelve subsystems, summarized by **S** (for 'subsystem') in the picture. Note that the *GrinCode* subsystem is split over the two products. This is not very logical, and a reason to merge the two product directories in the future. Furthermore, there are subdirectories for the two tools that are packed with the EHC project: *shuffle* and *ruler*. Their directories are summarized by **T** (for 'tool') in the picture. Finally, there is a separate directory for the *libutil* utilities.

The binary directory **bin** is structured as follows. There is a subdirectory for each view of the compiler (1 to 12, and some experimental ones). Remember that these views correspond with the subsequent extensions of the compiler. Each numbered subdirectory contains the executable products (*ehc*, *grini* and the obsolescent *grinc*) for that view. Also, both tools (*shuffle* and *ruler*) reside in this directory.

The intermediate directory **build** is most complex of all, but will seldomly be visited by human readers. It contains all the intermediate files resulting from the tool chains described in section 1.

The installation directory **install** contains a copy of the binary products once they are installed. Also, it contains the Cabal libraries (and their interface files) that can be used to separately compile parts of the system. Note that a separate library is build for each view/version combination.

All subdirectories are grouped in one super directory **EHC**. There is a copy of the entire tree in the *trunk*, and in each author's personal *branch*. From time to time we might freeze the trunk in a verison in directory *tags*. The trunk, each author's branch and each tagged version are collected in a main directory **ehc** which is stored in a public repository.

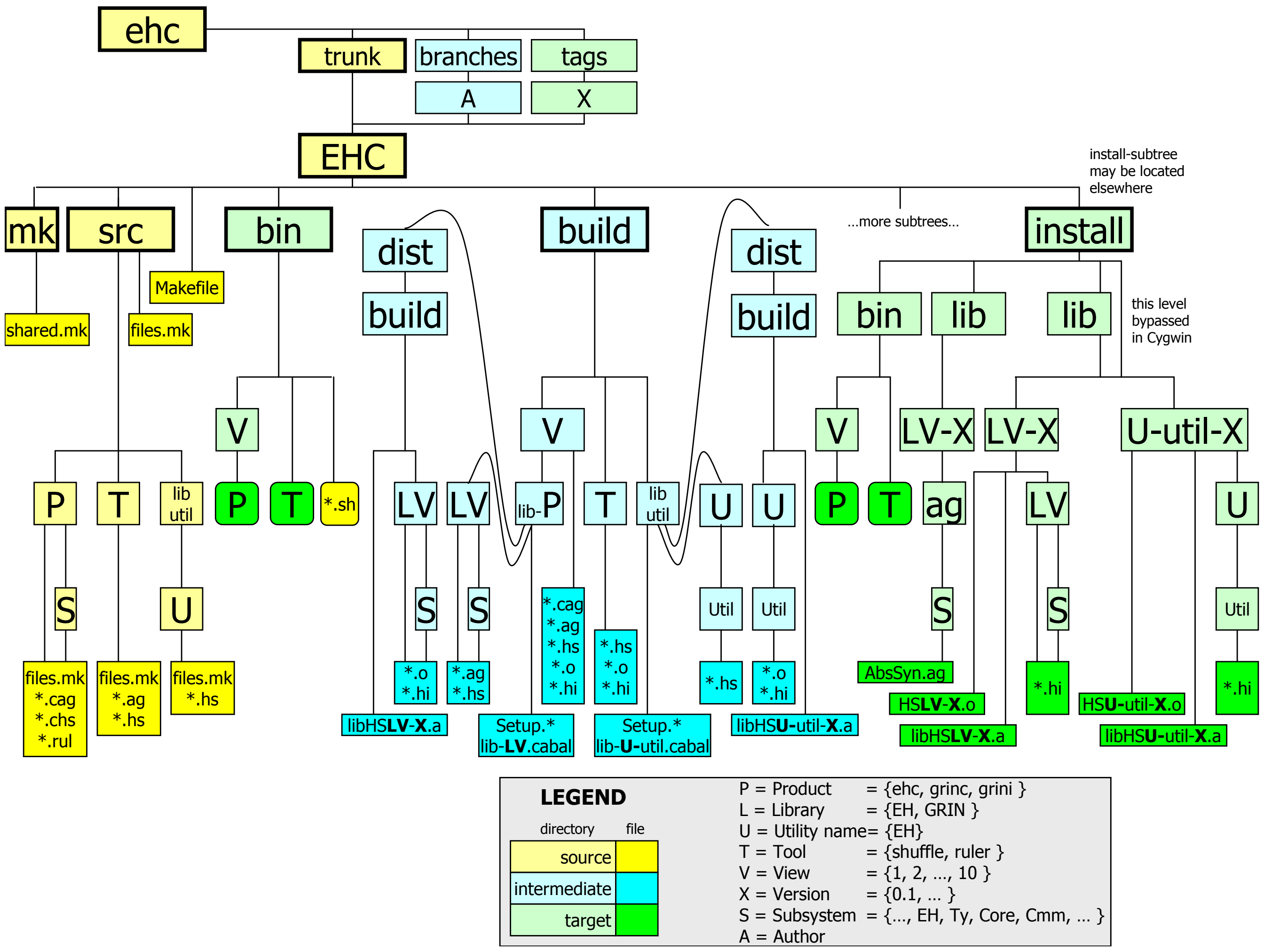

Figure 4: EHC file system layout