



Haskabelle – converting Haskell source files to Isabelle/HOL theories

Tobias Rittweiler, Florian Haftmann

16 April 2009

Abstract

This document gives an introduction to Haskabelle, an importer from Haskell source files to Isabelle/HOL theories.

Haskabelle

1.1 Introduction

1.1.1 What is Haskabelle?

Haskabelle is a converter from *Haskell* source files to *Isabelle/HOL* [1] theories implemented in *Haskell* itself.

1.1.2 Motivation

Isabelle/HOL can be regarded as a combination of a functional programming language and logic. Just like functional programming languages, it has its foundation in the typed lambda calculus, but is additionally crafted to allow the user to write arbitrary mathematical theorems in a structured and convenient way.

Haskell is a functional programming language that has succeeded in getting more and more momentum, not only in academia but increasingly also in industry. It is used for all kinds of programming tasks despite (or, perhaps, rather because) of its pureness, that is its complete lack of side-effects.

This pureness makes *Haskell* relate to *Isabelle/HOL* more closely than other functional languages. In fact, *Isabelle/HOL* can be considered a subset of *Haskell*.

Writing a converter from the convertible subset of *Haskell* to *Isabelle/HOL* seems thus like the obvious next step to facilitate machine-aided verification of *Haskell* programs. *Haskabelle* is exactly such a converter.

1.1.3 Implementation

There is one major design decision which users have to keep in mind. *Haskabelle* works on the Abstract Syntax Tree (AST) representation of *Haskell* programs exclusively. As a result, it is very restricted on what it knows about the validity of the program; for example, it does not perform type inference.

In fact, input source files are not checked at all beyond syntactic validity that is performed by the parser. Users are supposed to first run their *Haskell*

implementation of choice on the files to catch programming mistakes. In practice, this is not an impediment as it matches the putative workflow: *Haskabelle* is supposed to help the verification of already-written, or just-written programs.

Also, no proof checking is involved; that work is delegated to *Isabelle*. This means that only because the conversion seemingly succeeded, does not necessarily mean that *Isabelle* won't complain. A common example is that a *Haskell* function could be syntactically transformed to a corresponding *Isabelle/HOL* function, but *Isabelle* will refuse to accept it as it's not able to determine termination by itself.

Haskabelle performs its work in the following 5 phases.

Parsing

Each *Haskell* input file is parsed into an *Haskell* Abstract Syntax Tree representation. Additionally, module resolution is performed, i.e. the source files of the modules that the input files depend on are also read and parsed. So the actual output of this phase is a forest of *Haskell* ASTs.

Preprocessing

Each *Haskell* AST is normalised to a semantically equivalent but canonicalised form to simplify the subsequent converting phase. At the moment, the following transformations are performed:

- identifiers that would clash with reserved keywords or constants in *Isabelle/HOL* are renamed.
- pattern guards are transformed into nested if expressions.
- where clauses are transformed into let expressions.
- local function definitions are made global by renaming then uniquely.

Converting

After preprocessing, each *Haskell* AST consists entirely of toplevel definitions. Before the actual conversion, a dependency graph is generated for these toplevel definitions for two purposes: first to ensure that definitions appear textually before their uses; second to group mutually-recursive function

together. Both points are necessary to comply with requirements imposed by *Isabelle/HOL*.

Furthermore, a global environment is built in this phase that contains information about all identifiers, e.g. what they represent, in which module they belong to, whether they're exported, etc.

What *Haskell* language features are translated to which *Isabelle/HOL* constructs, is explained in section 1.3.

The output of this phase is a forest of *Isabelle/HOL* ASTs.

Adapting

While the previous phase converted the *Haskell* ASTs into their syntactically equivalent *Isabelle/HOL* ASTs, it has not attempted to map functions, operators, or algebraic data types, that preexist in *Haskell*, to their pedants in *Isabelle/HOL*. Such a mapping (or adaption) is performed in this phase.

Printing

The *Isabelle/HOL* ASTs are pretty-printed into an human-readable format so users can subsequently work with the resulting definitions, supply additional theorems, and verify their work.

1.2 Setup and usage

1.2.1 Prerequisites

We assume that the reader of this tutorial has some basic experience with *UNIX*, *Haskell*, and *Isabelle/HOL*.

Haskabelle is shipped in source code; this means you have to provide a working *Haskell* environment yourself, including some libraries. In order to make use of the theories generated by *Haskabelle*, you will also need an *Isabelle* release.

Haskell environment

The given version numbers just indicate which constellation has been tested – others might work, too.

First, the *Haskell* suite itself:

GHC Glasgow Haskell Compiler <http://www.haskell.org/ghc/> (version 6.10.1)

The following libraries are required:

mtl Monad transformer library.

<http://hackage.haskell.org/cgi-bin/hackage-scripts/package/mtl-1.1.0.1>

xml A simple XML library.

<http://hackage.haskell.org/cgi-bin/hackage-scripts/package/xml-1.3.3>

uniplate Uniform type generic traversals.

<http://hackage.haskell.org/cgi-bin/hackage-scripts/package/uniplate-1.2.0.3>

cpphs A liberalised re-implementation of cpp, the C pre-processor.

<http://hackage.haskell.org/cgi-bin/hackage-scripts/package/cpphs-1.6>

Happy Happy is a parser generator for Haskell.

<http://hackage.haskell.org/cgi-bin/hackage-scripts/package/happy-1.18.2>

The installation process provides a binary **happy** which must be accessible on your **PATH** to proceed!

haskell-src-ext Manipulating Haskell source: abstract syntax, lexer, parser, and pretty-printer.

<http://hackage.haskell.org/cgi-bin/hackage-scripts/package/haskell-src-extends-0.4.6>

Isabelle **release**

The latest *Isabelle* release is available from <http://isabelle.in.tum.de/download.html>.

Haskabelle **distribution**

The current *Haskabelle* release is available from <http://isabelle.in.tum.de/haskabelle.html> is tailored to the latest *Isabelle* release.

1.2.2 Basic usage

Understanding the distribution structure

Throughout this manual, qualified paths of executables on the shell prompt are relative to the root directory of the *Haskabelle* distribution.

Therein, among others, the following directories can be found:

bin/ Shell interfaces of *Haskabelle*
 doc/ Documentation
 default/ Default adaption files (see §1.4)
 ex/ Examples (see §1.5)

Converting theories

Haskabelle is invoked using the following command line:

```
bin/haskabelle <SRC1> .. <SRCn> <DST>
```

where <SRC1> ... <SRCn> is a list of *Haskell* source files to convert and <DST> is a directory to put the generated *Isabelle/HOL* theory files inside.

The *Prelude* theory the generated theory files depend on can be found in `default/Prelude.thy`.

Compiling

Haskabelle can be run directly from source; for efficient use it is recommended to build a binary from the sources, which is accomplished by invoking

```
bin/buildbin
```

1.3 A bluffer's glance at Haskabelle

In this section we want to provide a few examples to give the reader an impression of *Haskabelle*'s capabilities.

The following *Haskell* code represents a very simple interpreter:

```
module Example where

evalExp :: Exp -> Int
evalExp (Plus e1 e2) = evalExp e1 + evalExp e2
evalExp (Times e1 e2) = evalExp e1 * evalExp e2
evalExp (Cond b e1 e2)
  | evalBexp b = evalExp e1
  | otherwise = evalExp e2
evalExp (Val i) = i

evalBexp :: Bexp -> Bool
evalBexp (Equal e1 e2) = evalExp e1 == evalExp e2
evalBexp (Greater e1 e2) = evalExp e1 > evalExp e2
```

```

data Exp = Plus Exp Exp
         | Times Exp Exp
         | Cond Bexp Exp Exp
         | Val Int

data Bexp = Equal Exp Exp
         | Greater Exp Exp

```

Haskabelle will transform the above into the following:

```

theory Example
imports Prelude
begin

datatype Exp = Plus Exp Exp
            | Times Exp Exp
            | Cond Bexp Exp Exp
            | Val int
and Bexp = Equal Exp Exp
        | Greater Exp Exp

fun evalExp :: "Exp => int" and
    evalBexp :: "Bexp => bool"
where
  "evalExp (Plus e1 e2) = (evalExp e1 + evalExp e2)"
| "evalExp (Times e1 e2) = (evalExp e1 * evalExp e2)"
| "evalExp (Cond b e1 e2) = (if evalBexp b then evalExp e1
                             else evalExp e2)"
| "evalExp (Val i) = i"
| "evalBexp (Equal e1 e2) = heq (evalExp e1) (evalExp e2)"
| "evalBexp (Greater e1 e2) = (evalExp e1 > evalExp e2)"

end

```

We can note a couple of things at this point:

- The data type definitions have been moved before their uses.
- The two data type definitions have been chained together by an explicit `and` keyword. Likewise the function definitions have been grouped together. This stems from the mutual recursion inherent in the definitions.
- We use *Isabelle*'s function package.
- The pattern guards in `evalExp` have been transformed to an `if` expression.
- Preexisting *Haskell* functions and operators have been mapped to *Isabelle/HOL* counterparts.

- *Haskell* modules inherit from an implicit module `Prelude`; *Haskabelle* comes with a `Prelude.thy` which provides necessary context to cope with some *Haskell* features. We can see that an import of this the `Prelude` module is explicitly added by *Haskabelle*.
- The *Haskell* comparison operator `==` has been transformed to isatype-writer `heq` which is not defined by with *Isabelle/HOL* itself but within the `Prelude.thy` file. It names both an operator and a type class which has been constructed to match `==`, and *Haskell*'s type class `Eq`.

The next example illustrates a simple use of type classes.

```

module Classes where
class Monoid a where
  nothing :: a
  plus :: a -> a -> a

instance Monoid Integer where
  nothing = 0
  plus = (+)

-- prevent name clash with Prelude.sum
summ :: (Monoid a) => [a] -> a
summ [] = nothing
summ (x:xs) = plus x (summ xs)

class (Monoid a) => Group a where
  inverse :: a -> a

instance Group Integer where
  inverse = negate

sub :: (Group a) => a -> a -> a
sub a b = plus a (inverse b)

```

Haskabelle will transform this into the following:

```

theory Classes
imports Nats Prelude
begin
class Monoid = type +
  fixes nothing :: 'a
  fixes plus :: "'a => 'a => 'a"

instantiation int :: Monoid
begin
  definition nothing_int :: "int"
  where
    "nothing_int = 0"
  definition plus_int :: "int => int => int"
  where
    "plus_int = (op +)"
instance ..
end

```

```

fun summ :: "('a :: Monoid) list => ('a :: Monoid)"
where
  "summ Nil = nothing"
| "summ (x # xs) = plus x (summ xs)"

class Group = Monoid +
  fixes inverse :: "'a => 'a"

instantiation int :: Group
begin
  definition inverse_int :: "int => int"
  where
    "inverse_int = uminus"
instance ..
end

fun sub :: "('a :: Group) => ('a :: Group) => ('a :: Group)"
where
  "sub a b = plus a (inverse b)"

end

```

1.4 Adaption

1.4.1 The concept

Adaption allows to identify functions, types etc. from the *Haskell* source files with pre-existing counterparts in *Isabelle/HOL* by means of two mechanisms:

- An *adaption table* in a simple domain-specific language which specifies a table between identifiers of classes, types and functions in *Haskell* to their corresponding identifiers in *Isabelle/HOL*.
- A prelude theory containing a *Isabelle/HOL* base environment where *Haskabelle*'s output is supposed to be run implicitly within. By extending this, it is possible to adapt even more complex features of the *Haskell* programming language.

1.4.2 Setting up your own adaption

Haskabelle provides some default adaptions already in directory `default`. You can setup your own adaption according to the following steps:

Copy default

Typically you will want to use the default adaption as a starting point, so copy the `default` directory to a directory of your choice (which we will refer to as `<ADAPT>`).

Adapt the prelude theory

If desired, adapt the prelude theory `<ADAPT>/Prelude.thy`.

Edit adaption table

The adaptations themselves reside in `<ADAPT>/adapt.txt` and can be edited there.

Process adaptations

To make the adaptations accessible to *Haskabelle*, execute the following:

```
bin/mk_adapt <ADAPT>
```

This also includes some basic consistency checking.

If you have multiple *Isabelle* versions on your machine, you can select one particular by setting the shell variable `ISABELLE_PROCESS` (usually `ISABELLE_HOME/bin/isabelle-process`) to the process wrapper of the desired *Isabelle*.

Use this adaption during conversion

A particular adaption other than default is selected using the `--adapt` command line switch:

```
bin/haskabelle --adapt <ADAPT> <SRC1> .. <SRCn> <DST>
```

1.5 Examples

Examples for *Haskabelle* can be found in the `ex/src.hs` directory in the distribution. They can be converted at a glance using the following command:

```
bin/regression
```

Each generated theory then is re-imported into *Isabelle*. If you have multiple *Isabelle* versions on your machine, you can select one particular by setting the shell variable `ISABELLE_TOOL` (usually `ISABELLE_HOME/bin/isabelle`) to the tool wrapper of the desired *Isabelle*.

Bibliography

- [1] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002. LNCS Tutorial 2283.