Implementation of a Pragmatic Translation from Haskell into Isabelle/HOL

Patrick Bahr
pa-ba@arcor.de

NICTA Sydney, TU Wien

December 17, 2008
Outline

1 Introduction
   - Haskell vs. Isabelle/HOL
   - Motivation
   - Goals

2 Translating Haskell into Isabelle/HOL
   - Haskell vs. Isabelle/HOL
   - Implementation

3 Conclusions
Outline

1 Introduction
   • Haskell vs. Isabelle/HOL
   • Motivation
   • Goals

2 Translating Haskell into Isabelle/HOL
   • Haskell vs. Isabelle/HOL
   • Implementation

3 Conclusions
# Haskell vs. Isabelle/HOL

## Haskell in a nutshell

- **purely functional** programming language
- **non-strict** semantics (mostly implemented by lazy evaluation)
- comprehensive type system: Hindley-Milner (restricted $F_\omega$) + type classes
- uses monads to allow side effects

## Isabelle/HOL in a nutshell

- Isabelle: generic theorem prover
- HOL: Isabelle formulation of classical higher-order logic
- based on simply typed lambda calculus (system $F_1$) $\Rightarrow$ comparatively weak type system extended with type classes

---

Patrick Bahr (NICTA Sydney, TU Wien)  
Translating Haskell into Isabelle/HOL  
December 17, 2008
Haskell vs. Isabelle/HOL

Haskell in a nutshell
- purely functional programming language
- non-strict semantics (mostly implemented by lazy evaluation)
- comprehensive type system: Hindley-Milner (restricted $F_\omega$) + type classes
- uses monads to allow side effects

Isabelle/HOL in a nutshell
- Isabelle: generic theorem prover
- HOL: Isabelle formulation of classical higher-order logic
- based on simply typed lambda calculus (system $F_1$) $\Rightarrow$ comparatively weak type system
- extended with type classes
Haskell vs. Isabelle/HOL

Haskell in a nutshell

- purely functional programming language
- non-strict semantics (mostly implemented by lazy evaluation)
- comprehensive type system: Hindley-Milner (restricted $F_\omega$) + type classes
- uses monads to allow side effects

Isabelle/HOL in a nutshell

- Isabelle: generic theorem prover
- HOL: Isabelle formulation of classical higher-order logic
- based on simply typed lambda calculus (system $F_1$)
  $\rightsquigarrow$ comparatively weak type system
- extended with type classes

more details when we come to the implementation
Motivation

Program verification

- Haskell’s semantics allows comparatively easy reasoning
- there is no theorem prover for Haskell!
  \[\mapsto\] translate Haskell into language of a generic theorem prover

Example: l4.verified project
aim: formalisation and verification of a microkernel prototype implementation in Haskell
translation into Isabelle/HOL
\[\mapsto\] executable model
reasoning about executable model in Isabelle/HOL
Motivation

Program verification
- Haskell’s semantics allows comparatively easy reasoning
- there is no theorem prover for Haskell!
  ⇝ translate Haskell into language of a generic theorem prover

Example: l4.verified project
- aim: formalisation and verification of a microkernel
- prototype implementation in Haskell
- translation into Isabelle/HOL ⇝ executable model
- reasoning about executable model in Isabelle/HOL
Goals

- cover a large subset of Haskell’s syntax
- result should be easily readable
  - preserve syntactic structure as much as possible
  - translate syntactic sugar as well
- keep reasoning simple $\leadsto$ Isabelle/HOL
Goals

- cover a large subset of Haskell’s syntax
- result should be easily readable
  - preserve syntactic structure as much as possible
  - translate syntactic sugar as well
- keep reasoning simple $\leadsto$ Isabelle/HOL

$\leadsto$ Translation is neither sound nor complete!
Goals

- cover a large subset of Haskell’s syntax
- result should be easily readable
  - preserve syntactic structure as much as possible
  - translate syntactic sugar as well
- keep reasoning simple $\leadsto$ Isabelle/HOL

$\leadsto$ Translation is neither sound nor complete!

Implementation

- implementation language: Haskell
- based on existing work from TU Munich
Outline

1. Introduction
   - Haskell vs. Isabelle/HOL
   - Motivation
   - Goals

2. Translating Haskell into Isabelle/HOL
   - Haskell vs. Isabelle/HOL
   - Implementation

3. Conclusions
Haskell vs. Isabelle/HOL – Non-strictness/Partiality

- In Isabelle/HOL only total functions are definable
  \[ \leadsto \text{recursive definitions need termination proof} \]
- Haskell is Turing-complete \[ \leadsto \text{partial functions definable} \]
- Haskell’s semantics is non-strict

Example (Haskell)

\[
\text{from} :: \text{Int} \rightarrow [\text{Int}]
\text{from} n = n : \text{from} (n + 1)
\]

\text{from} does not terminate for any input
\[ \leadsto \text{not definable in Isabelle/HOL} \]

Due to non-strictness this function is still usable in Haskell

Example (Haskell)

\[
\text{nPrimes} :: \text{Int} \rightarrow [\text{Int}]
\text{nPrimes} n = \text{take} n (\text{filter} \text{isPrime} (\text{from} 1))
\]

Definitions that depend on non-strictness have to be avoided!
Haskell vs. Isabelle/HOL – Non-strictness/Partiality

- In Isabelle/HOL only total functions are definable
  ⟷ recursive definitions need termination proof
- Haskell is Turing-complete ⟷ partial functions definable
- Haskell’s semantics is non-strict

Example (Haskell)

```haskell
from :: Int -> [Int]
from n = n : from (n+1)
```

Definitions that depend on non-strictness have to be avoided!
Haskell vs. Isabelle/HOL – Non-strictness/Partiality

- In Isabelle/HOL only total functions are definable
  - Recursive definitions need termination proof
- Haskell is Turing-complete \( \rightsquigarrow \) partial functions definable
- Haskell’s semantics is non-strict

Example (Haskell)

```haskell
from :: Int -> [Int]
from n = n : from (n+1)
```

- From does not terminate for any input
  - Not definable in Isabelle/HOL
- Due to non-strictness this function is still usable in Haskell
Haskell vs. Isabelle/HOL – Non-strictness/Partiality

- in Isabelle/HOL only total functions are definable
  - recursive definitions need termination proof
- Haskell is Turing-complete \(\Rightarrow\) partial functions definable
- Haskell’s semantics is non-strict

**Example (Haskell)**

```haskell
from :: Int -> [Int]
from n = n : from (n+1)
```

- `from` does not terminate for any input
  - \(\Rightarrow\) not definable in Isabelle/HOL
- due to non-strictness this function is still usable in Haskell

**Example (Haskell)**

```haskell
nPrimes :: Int -> [Int]
nPrimes n = take n (filter isPrime (from 1))
```
Haskell vs. Isabelle/HOL – Non-strictness/Partiality

- In Isabelle/HOL, only total functions are definable.
  - Recursive definitions need termination proof.
- Haskell is Turing-complete, so partial functions are definable.
- Haskell’s semantics is non-strict.

**Example (Haskell)**

```haskell
from :: Int -> [Int]
from n = n : from (n + 1)
```

- `from` does not terminate for any input.
  - Not definable in Isabelle/HOL.
- Due to non-strictness, this function is still usable in Haskell.

**Example (Haskell)**

```haskell
nPrimes :: Int -> [Int]
nPrimes n = take n (filter isPrime (from 1))
```

Definitions that depend on non-strictness have to be avoided!
Haskell vs. Isabelle/HOL – Local Function Definitions

- **Haskell** allows recursive function definitions in local contexts (using `let` or `where`)
- in **Isabelle/HOL** recursive function definitions are only allowed at the top level

Example (Haskell)

```haskell
sumLen :: Int -> [a] -> [a] -> Int
sumLen s l1 l2 =
  let len [] = 0
      len (x:xs) = len xs + s
  in len l1 + len l2
```
Haskell vs. Isabelle/HOL – Local Function Definitions

- Haskell allows recursive function definitions in local contexts (using `let` or `where`)
- In Isabelle/HOL, recursive function definitions are only allowed at the top level

Example (Haskell)

```
sumLen :: Int -> [a] -> [a] -> Int
sumLen s l1 l2 = let len [] = 0
                 len (x:xs) = len xs + s
                 in len l1 + len l2
```
Haskell vs. Isabelle/HOL – Local Function Definitions II

- local function definitions have to be moved to the top level
- closures have to be made explicit

Example (Isabelle/HOL)

```haskell
fun len1 where
  "len1 _ Nil = 0"
| "len1 s (x # xs) = len1 s xs + s"

fun sumLen :: "int => 'a list => 'a list => int"
where
  "sumLen s l1 l2 = ( let len = len1 s
                      in len l1 + len l2 )"
```

Our implementation is able to make these transformations!
local function definitions have to be moved to the top level
- closures have to be made explicit

Example (Isabelle/HOL)

```haskell
fun len1
where
  "len1 _ Nil = 0"
| "len1 s (x # xs) = len1 s xs + s"

fun sumLen :: "int => 'a list => 'a list => int"
where
  "sumLen s l1 l2 = (let len = len1 s
                   in len l1 + len l2)"
```
Haskell vs. Isabelle/HOL – Local Function Definitions II

- local function definitions have to be moved to the top level
- closures have to be made explicit

Example (Isabelle/HOL)

```haskell
fun len1
where
  "len1 _ Nil = 0"
| "len1 s (x # xs) = len1 s xs + s"

fun sumLen :: "int => 'a list => 'a list => int"
where
  "sumLen s l1 l2 = (let len = len1 s
                    in len l1 + len l2)"
```

Our implementation is able to make these transformations!
Haskell vs. Isabelle/HOL – Order of Definitions

- In Haskell definitions can appear in any order.
- In Isabelle/HOL:
  - An identifier has to be defined before usage.
  - Mutual recursive definitions have to be made in parallel.
Haskell vs. Isabelle/HOL – Order of Definitions

- in Haskell definitions can appear in any order
- in Isabelle/HOL:
  - an identifier has to be defined before usage
  - mutual recursive definitions have to be made in parallel

Our implementation reorders definitions accordingly!
Haskell vs. Isabelle/HOL – Polymorphism

- **Haskell**: polymorphism over type constructors (of arbitrary kind)
- **Isabelle/HOL**: polymorphism over types only

**Example (type constructors)**

- types (constructors of kind \(\ast\)): \(\text{Int}\), \([\text{Bool}]\), \(\text{Int} \to \text{Bool}\), ...  
- type constructors of first-order kind: list (\(\text{[]}: \ast \to \ast\)), sum (\(\text{Either}: \ast \to (\ast \to \ast)\))  
- type constructor of higher-order kind: Tree: \((\ast \to \ast) \to (\ast \to \ast)\)

\[
\text{data Tree } c \; a = \text{Node } a \; (c \; \text{Tree } c \; a)
\]
Haskell vs. Isabelle/HOL – Ad Hoc Polymorphism

- **Haskell**: type classes + constructor classes
- **Isabelle/HOL**: type classes only
Haskell vs. Isabelle/HOL – Ad Hoc Polymorphism

- **Haskell**: type classes + constructor classes
- **Isabelle/HOL**: type classes only

**Example (classes)**

- **type class**:

  ```haskell
  class (Eq a, Show a) => Num a where
  (+), (-), (*) :: a -> a -> a
  negate :: a -> a
  : :
  ```
Haskell vs. Isabelle/HOL – Ad Hoc Polymorphism

- Haskell: type classes + constructor classes
- Isabelle/HOL: type classes only

Example (classes)

- type class:

```
class (Eq a, Show a) => Num a where
  (+), (-), (*) :: a -> a -> a
  negate :: a -> a
```

- constructor class:

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
```
- monad class is **not definable** in Isabelle/HOL!
- monads are crucial for practical Haskell programs
- monads can be used to describe computations with side effects
Haskell vs. Isabelle/HOL – Ad Hoc Polymorphism II

- monad class is not definable in Isabelle/HOL!
- monads are crucial for practical Haskell programs
- monads can be used to describe computations with side effects

Our solution

- Translate only instances of the class Monad!
- each monad instance has to use different names for the operation
e.g. one monad uses >>=, return; another one uses >>=', return'
- type inference has to be performed to rename the operations correctly
- not full type inference is used, only a simple heuristics
Further things that are taken care of in the translation

- as-patterns
Haskell vs. Isabelle/HOL – Misc.

Further things that are taken care of in the translation

- as-patterns

Example

In Haskell:

```haskell
f :: [Int] -> [Int]
f l@(_:_) = 0 : l
f l@(\[\]) = 1 : l
```

In Isabelle/HOL:

```isabelle
fun f where
  "f (a0 # a1) = (let l = (a0 # a1) in 0 # l)"
| "f Nil = (let l = Nil in 1 # l)"
```
Haskell vs. Isabelle/HOL – Misc.

Further things that are taken care of in the translation

- as-patterns

Example

In Haskell:

```haskell
f :: [Int] -> [Int]
f l@(\_:\_) = 0 : l
f l@[[]] = 1 : l
```

In Isabelle/HOL:

```isabelle
fun f where
  "f (a0 # a1) = (let l = (a0 # a1)
      in 0 # l)"
| "f Nil = (let l = Nil
      in 1 # l)"
```

Patrick Bahr (NICTA Sydney, TU Wien)
Further things that are taken care of in the translation

- as-patterns
- labelled fields in data types
Haskell vs. Isabelle/HOL – Misc.

Further things that are taken care of in the translation

- as-patterns
- labelled fields in data types

Example (Haskell)

```haskell
data MyRecord = A { aField1 :: String, 
                     common1 :: Bool, 
                     common2 :: Int } 
   | B { bField1 :: Bool, 
        bField2 :: Int, 
        common1 :: Bool, 
        common2 :: Int } 
   | C Bool Int String
```
Further things that are taken care of in the translation

- as-patterns
- labelled fields in data types

Example (Haskell)

```haskell
data MyRecord = A { aField1 :: String, common1 :: Bool, common2 :: Int } |
                B { bField1 :: Bool, bField2 :: Int, common1 :: Bool, common2 :: Int } |
                C Bool Int String
```

\[\rightarrow\] This is reduced to an ordinary data type!
Haskell vs. Isabelle/HOL – Misc.

Further things that are taken care of in the translation

- as-patterns
- labelled fields in data types
- guards
Further things that are taken care of in the translation

- as-patterns
- labelled fields in data types
- guards

Example (Haskell)

```
insert :: Int -> [Int] -> [Int]
insert n [] = [n]
insert n (m:ms)
  | n < m     = n:m:ms
  | otherwise = m: insert n ms
```
Haskell vs. Isabelle/HOL – Misc.

Further things that are taken care of in the translation

- as-patterns
- labelled fields in data types
- guards

Example (Haskell)

```haskell
insert :: Int -> [Int] -> [Int]
insert n [] = [n]
insert n (m:ms)
  | n < m    = n:m:ms
  | otherwise = m: insert n ms
```

⇝ Guards are reduced to if-then-else expressions!
Overall Design of Implementation – Parsing

- Parse each Haskell module to a syntax tree.
- Imported modules are located and parsed as well.
- Parser only verifies context-free part of the syntax.
- Syntactically correct Haskell program is assumed.

1. Parsing

Haskell modules

Haskell syntax trees
Guards are transformed into if-then-else expressions.

Local function definitions are transformed into top-level function definitions.

Keywords and identifiers defined in the Isabelle/HOL library are renamed.
Overall Design of Implementation – Analysis

Some global information about the program is collected:
- type annotations
- the module where an identifier was defined
- what an identifier refers to (type, function etc.)
- associativity and precedence of defined operators
Definitions are reordered according to their dependencies.

Haskell syntax trees are translated into Isabelle/HOL syntax trees.

1. Parsing
   - Haskell modules
   - Haskell syntax trees

2. Preprocessing
   - Haskell syntax trees (simplified)

3. Analysis
   - Context
   - Information
   - Isabelle syntax trees (intermediate)

4. Conversion
Renaming of predefined identifiers, e.g.:

- `Int` $\mapsto$ `int`
- `[]` $\mapsto$ `Nil`
- `++` $\mapsto$ `@`
Isabelle/HOL syntax trees are written into theory files.

- Parsing
- Preprocessing
- Conversion
- Adaptation
- Printing
- Analysis

Haskell modules

1. Parsing
   Haskell syntax trees

2. Preprocessing
   Haskell syntax trees (simplified)

3. Analysis
   Context
   Information

4. Conversion
   Isabelle syntax trees (intermediate)

5. Adaptation
   Isabelle syntax trees

6. Printing
   Isabelle theories
Outline

1 Introduction
   - Haskell vs. Isabelle/HOL
   - Motivation
   - Goals

2 Translating Haskell into Isabelle/HOL
   - Haskell vs. Isabelle/HOL
   - Implementation

3 Conclusions
Summary

Original implementation covered

- case, if-then-else, and let expressions
- list comprehensions
- where bindings
- as-patterns
- guards
- mutually recursive functions and data type definitions
- simple pattern bindings
- definitions and instantiations of type classes
## Summary

### Original implementation covered

- case, if-then-else, and let expressions
- list comprehensions
- where bindings
- as-patterns
- guards
- mutually recursive functions and data type definitions
- simple pattern bindings
- definitions and instantiations of type classes

Some parts of the translations were unsound!
### Our Contributions

- ✔ mutually recursive function and data type definitions
- ✔ as-patterns
- ✔ guards
  - data types with labelled fields
  - closures in local function definitions
  - monomorphic uses of monads

---

What is missing:
- constructor type classes
- ⇝ polymorphic uses of monads
- non-simple pattern bindings
- irrefutable patterns
Our Contributions

- mutually recursive function and data type definitions
- as-patterns
- guards
  - data types with labelled fields
  - closures in local function definitions
  - monomorphic uses of monads

What is missing

- constructor type classes $\sim$ polymorphic uses of monads
- non-simple pattern bindings
- irrefutable patterns
Conclusions

What do we have

- translation is unsound!
- most of the Haskell 98 language can be translated
- resulting Isabelle/HOL formalisation is close to Haskell program
- comparatively easy reasoning in Isabelle/HOL
- adequate translation for most purposes $\rightsquigarrow$ l4.verified
Conclusions

What do we have

- translation is unsound!
- most of the Haskell 98 language can be translated
- resulting Isabelle/HOL formalisation is close to Haskell program
- comparatively easy reasoning in Isabelle/HOL
- adequate translation for most purposes $\implies$ l4.verified

Alternative Approach

- logic HOLCF is well suited to formalise partiality and non-strictness
- even constructor classes can be formalised
- reasoning in Isabelle/HOLCF is more complicated
Coping with Large Data Types

Dealing with syntax trees $\Rightarrow$ dealing with large data types.

Data Types Defining Haskell Syntax Trees

- 500 lines of Haskell code
- 51 data types
- “largest” data type contains 45 constructors
Coping with Large Data Types

Dealing with syntax trees $\Rightarrow$ dealing with large data types.

Data Types Defining Haskell Syntax Trees

- 500 lines of Haskell code
- 51 data types
- “largest” data type contains 45 constructors

You don’t want to write all the code for all those data types and each of their constructors!

If you have to write it you only want to write it once!
Coping with Large Data Types

Dealing with syntax trees ⇒ dealing with large data types.

Data Types Defining Haskell Syntax Trees

- 500 lines of Haskell code
- 51 data types
- “largest” data type contains 45 constructors

- You don’t want to write all the code for all those data types and each of their constructors!
  ⇒ Generic Programming + Code Generation

- If you have to write it you only want to write it once!
  ⇒ Modularity
Generic Programming
“Scrap Your Boilerplate”

Problem Addressed by SYB

- traverse a data structure to transform or query it
- only a few parts of the data structure are relevant

Example
compute free variables of an expression
transform where clauses into let expressions
Problem Addressed by SYB

- traverse a data structure to transform or query it
- only a few parts of the data structure are relevant

Example

- compute free variables of an expression
- transform where clauses into let expressions
Problem Addressed by SYB
- traverse a data structure to transform or query it
- only a few parts of the data structure are relevant

Example
- compute free variables of an expression
- transform \textit{where} clauses into \textit{let} expressions

Difficulties when Applying SYB in our Setting
- often \textit{context information} is necessary
- We want to define a piece of context information \textit{only once}. 
Environments

Data Structure as a Tree

- Changes environment

A and B needed
A and C needed
Environments

Data Structure as a Tree

A
B
C
= changes environment
A and B needed
A and C needed

Defining Environments by $a \rightarrow (e \rightarrow e)$
- $a$ is the type of the current node
- $e$ is the type of the environment
Extending SYB by Environment Propagation

Extension to SYB

- allows to define environments
- allows to combine environments
- provides traversal strategies with environment propagation
Extending SYB by Environment Propagation

Extension to SYB
- allows to define environments
- allows to combine environments
- provides traversal strategies with environment propagation

Generalisation of Environment Propagation
- non-uniform propagation
- monadic computations to define an environment