Implementation of a Pragmatic Translation from Haskell into Isabelle/HOL

Patrick Bahr
pa-ba@arcor.de

NICTA

October 29, 2008
Outline

1. Introduction
2. Existing Implementation
3. Extensions to the Implementation
   - Translating Further Language Features
   - Useful Techniques
4. Summary
Outline

1 Introduction

2 Existing Implementation

3 Extensions to the Implementation
   - Translating Further Language Features
   - Useful Techniques

4 Summary
Motivation

seL4
- Prototype implementation in Haskell.
- Executable model in Isabelle/HOL for verification.
Motivation

seL4

- Prototype implementation in Haskell.
- Executable model in Isabelle/HOL for verification.

No Theorem Prover for Haskell

- Haskell allows easy reasoning about its semantics.
- no theorem prover to automate this
Isabelle/HOL as Target Language

Benefits

- automated translation is simpler
- resulting translation is close to original Haskell code
- reasoning in HOL is easier (than in HOLCF)

Drawbacks

- translation is not complete
- translation is not sound

issues:

▶ comprehensive language features (e.g. type system)
▶ non-strictness
<table>
<thead>
<tr>
<th>Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td>automated translation is simpler</td>
</tr>
<tr>
<td>resulting translation is close to original Haskell code</td>
</tr>
<tr>
<td>reasoning in HOL is easier (than in HOLCF)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Drawbacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>translation is <strong>not</strong> complete</td>
</tr>
<tr>
<td>translation is <strong>not</strong> sound</td>
</tr>
<tr>
<td>issues:</td>
</tr>
<tr>
<td>▶ comprehensive language features (e.g. type system)</td>
</tr>
<tr>
<td>▶ non-strictness</td>
</tr>
</tbody>
</table>
Outline

1. Introduction

2. Existing Implementation

3. Extensions to the Implementation
   - Translating Further Language Features
   - Useful Techniques

4. Summary
Design of the Implementation

The translation is performed in six steps:

- Parsing
- Preprocessing
- Analysis
- Conversion
- Adaption
- Printing
Design of the Implementation

The translation is performed in six steps:

- Parsing
- Preprocessing
- Analysis
- Conversion
- Adaption
- Printing
Preprocessing

- **Guards** are transformed into *if-then-else* expressions.
- **Local function definitions** are transformed into top-level function definitions.
- **As-patters** are transformed into additional nested pattern matches.
- **Keywords** and identifiers defined in the Isabelle/HOL library are renamed.
Conversion

- Definitions are reordered according to their dependencies.
- Haskell syntax trees are translated into Isabelle/HOL syntax trees.

<table>
<thead>
<tr>
<th>Translation in a (Very Small) Nutshell</th>
</tr>
</thead>
<tbody>
<tr>
<td>function bindings</td>
</tr>
<tr>
<td>simple pattern bindings</td>
</tr>
<tr>
<td>data type declarations</td>
</tr>
<tr>
<td>type class declararions</td>
</tr>
<tr>
<td>instance declarations</td>
</tr>
</tbody>
</table>
Issues of the Implementation

Things that are not Supported

- data types with field labels
- closures of local function definitions
- constructor type classes (+ multi-parameter type classes)
- irrefutable patterns

Things that Go Wrong

- Dependencies on data types are ignored.
- The translation of as-patterns is unsound.
Outline

1. Introduction

2. Existing Implementation

3. Extensions to the Implementation
   - Translating Further Language Features
   - Useful Techniques

4. Summary
Our Contributions

- translation of data types with labelled fields
- translation of closures
- heuristic to translate monadic programs
- infrastructure to customise the translation
- dependencies on type definitions are respected
- sound translation of as-patterns*
- testing framework
Our Contributions

- translation of data types with labelled fields
- translation of closures
- heuristic to translate monadic programs
- infrastructure to customise the translation
- dependencies on type definitions are respected
- sound translation of as-patterns*
- testing framework
Data Types with Labelled Fields

Haskell

data MyRecord = A { aField1 :: Int,
   aField2 :: String,
   common :: Char }
| B { bField1 :: Bool,
   bField2 :: Int,
   bField3 :: Int,
   common :: Char }
| C Bool Bool String
Data Types with Labelled Fields

Haskell

```haskell
data MyRecord = A { aField1 :: Int,
                    aField2 :: String,
                    common :: Char }
               | B { bField1 :: Bool,
                    bField2 :: Int,
                    bField3 :: Int,
                    common :: Char }
               | C Bool Bool String
```

Isabelle/HOL

```isabelle
datatype MyRecord = A int string char
                  | B bool int int char
                  | C bool bool bool string
```
Fields as Selection Functions

```plaintext
primrec aField1 :: "MyRecord => int"
  where
    "aField1 (A x _ _) = x"

primrec common :: "MyRecord => char"
  where
    "common (B _ _ _ x) = x"
    | "common (A _ _ x) = x"

: 
```
Haskell

```haskell
constr :: MyRecord
constr = A{ aField1 = 1, common = '2'}
```
Haskell

\[
\text{constr :: MyRecord} \\
\text{constr = A\{ aField1 = 1, common = '2'\}}
\]

Isabelle/HOL

definition constr :: "MyRecord"
where
  "constr = A 1 arbitrary CHR '2'"
Related Syntax

Updates

Haskell

```haskell
update :: MyRecord -> MyRecord
update x = x{aField2 = "foo"}
```
Related Syntax

Updates

Haskell

```haskell
update :: MyRecord -> MyRecord
update x = x{aField2 = "foo"}
```

Isabelle/HOL

```isabelle
fun update :: "MyRecord => MyRecord"
where
  "update x = (case x of
    A v1 v2 v3
    => A v1 "'foo'" v3
    | _ => arbitrary)"
```
Related Syntax
Pattern Matching

Haskell

```haskell
pattern :: MyRecord -> Int
pattern A{aField1 = val} = val
pattern B{bField3 = val} = val
pattern (C v1 v2 v3) = 1
```
Related Syntax
Pattern Matching

Haskell

pattern :: MyRecord -> Int
pattern A{aField1 = val} = val
pattern B{bField3 = val} = val
pattern (C v1 v2 v3) = 1

Isabelle/HOL

fun pattern :: "MyRecord => int"
where
  "pattern A val _ _ = val"
| "pattern B _ _ val _ = val"
| "pattern (C v1 v2 v3) = 1"
functions can be defined locally using where and let
transformed to top-level definitions
Closures

- Functions can be defined \textit{locally} using \texttt{where} and \texttt{let}
- Transformed to top-level definitions

But

- Locally defined function can refer to \texttt{free variables} only bound in the local context.
  \Rightarrow\ Closure
- The transformation has to make the \texttt{environment} of the closure explicit.
An Example

Haskell Definition of Several Closures

\[
\text{func } x \ y = \text{sum } x + \text{addToX } y \quad -- \text{ closure:}
\]
\[
\text{where } \text{addToX } y = x + y \quad -- \ x
\]
\[
\text{addToY } x = x + y \quad -- \ y \ (+ \ x)
\]
\[
w = \text{addToY } x
\]
\[
\text{sum } y = w + y \quad -- \ x \ (+ \ y)
\]
An Example

Haskell Definition of Several Closures

```
func x y = sum x + addToX y -- closure:
  where addToX y = x + y -- x
      addToY x = x + y -- y (+ x)
      w = addToY x
      sum y = w + y -- x (+ y)
```

Transformed Top-level Definitions

```
addToX' x y = x + y
addToY' (_, y) x = x + y
sum' env y = let (x, _) = env
       w = addToY' env x
       in w + y
```
An Example

The Final Result

```haskell
addToX' x y = x + y
addToY' (_, y) x = x + y
sum' env y = let (x, _) = env
             w = addToY' env x
             in w + y
```

Patrick Bahr (NICTA)
Translating Haskell into Isabelle/HOL
October 29, 2008 20 / 29
An Example

The Final Result

\begin{verbatim}
addToX' x y = x + y
addToY' (_, y) x = x + y
sum' env y = let (x, _) = env
    w = addToY' env x
    in w + y

func x y = let addToX = addToX' x
           addToY = addToY' (x, y)
           sum = sum' (x, y)
           w = addToY x
    in sum x + addToX y
\end{verbatim}
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
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!

If you have to write it you only want to write it once!
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!
  $\Rightarrow$ Generic Programming + Code Generation

- If you have to write it you only want to write it once!
  $\Rightarrow$ Modularity
QuickCheck

- allows to specify and test algebraic properties
- needs generators that produce random test data
- tests properties by generating a value for each universally quantified element
- uses type system to get the right generator for each type
Testing with QuickCheck

QuickCheck

- allows to specify and test algebraic properties
- needs generators that produce random test data
- tests properties by generating a value for each universally quantified element
- uses type system to get the right generator for each type

We have to implement test data generators for Haskell syntax trees!
QuickCheck

- allows to specify and test algebraic properties
- needs generators that produce random test data
- tests properties by generating a value for each universally quantified element
- uses type system to get the right generator for each type

We have to implement test data generators for Haskell syntax trees!

Generators for Data Types

- randomly choose a constructor,
- generate values for the argument of the constructor, and
- combine the results
Template Haskell

Extension to Haskell that allows to generate Haskell code at compile time.
Template Haskell

Extension to Haskell that allows to generate Haskell code at compile time.

Using Template Haskell to Define Test Data Generators
We implemented a library of Template Haskell functions that allow
- to define most generators in one line, and
- to customise the defined generators.
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 \texttt{where} clauses into \texttt{let} expressions
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

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

Data Structure as a Tree

A

B

C

A and B needed

A and C needed

= changes environment

A -> (e -> e)

a is the type of the current node

e is the type of the environment

Patrick Bahr (NICTA)

Translating Haskell into Isabelle/HOL

October 29, 2008 25 / 29
Environments

Data Structure as a Tree

= 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
Outline

1. Introduction

2. Existing Implementation

3. Extensions to the Implementation
   - Translating Further Language Features
   - Useful Techniques

4. Summary
Done

- eliminated most shortcomings of the previous implementation
- customisation mechanism
- testing framework
Summary

Done

- eliminated most shortcomings of the previous implementation
- customisation mechanism
- testing framework

Loose Ends

- circular dependencies between modules
- applying the translation to seL4.
Thank you!