Bringing Isabelle/HOLCF Closer to Haskell

Brian Huffman

November 16, 2005

Haskell-to-Isabelle Translator

- Intended to be a light-weight translation:
  - Translators are trusted code; they should be small and simple!
  - Semantics should be encoded in the theorem prover, not in the translator
  - If the theorem prover supports all the features of the source language, then the translator just maps syntax
- What to do if the theorem prover lacks support for a language feature?
  - Have translator convert it to simpler language features
  - Disallow source programs that use the feature
  - Extend the theorem prover to support the feature

Haskell-to-Isabelle Translator

- Directly supported features:
  - simple datatype declarations
  - simple case expressions
  - pattern-matching function definitions
- Translated features:
  - function and datatype dependencies
- Soon-to-be-allowed features:
  - full case expressions with nested patterns
  - local function and value definitions
  - datatypes with indirect recursion

Syntax Translations in Isabelle

- In Isabelle, there is usually a close connection between syntax and semantics...
- ...but fancier syntax can be implemented with syntax translations
- Syntax translations, or macros, are simply rewrite rules
  - The macros rewrite Isabelle's abstract syntax trees
  - One set of macros is applied during parsing
  - Another set is applied during pretty printing

Syntax Translations in Isabelle

- Example: split function from Isabelle/HOL

```
consts
  split :: "('a => 'b => 'c) => 'a * 'b => 'c"
translations
  "λ(x,y,zs).b" == "split(λx. λ(y,zs).b)"
  "λ(x,y).b" == "split(λx. λy. b)"
```

- Each translation is really two macros:
  - Left-to-right is a parse macro, right-to-left is a print macro
- Macros must have linear patterns, cannot introduce new variables

Current HOLCF Pattern Matching

HOLCF 2005 supports two simple forms of pattern matching:

- Lambda abstractions can match against tuples

```
translations "LAM <x,y>. t" == "csplit (LAM x y. t)"
```

- Types defined by the domain package get a case analysis combinator

```
domain 'a maybe = Nothing | Just (lazy 'a)
consts
  maybe_when :: 'a -> ('b -> 'a) -> 'b maybe -> 'a
translations
  "case t of Nothing => x | Just a => y"
      == "maybe_when x (LAM a. y) t"
```

New HOLCF Pattern Matching

Design requirements:

- Must support abitrarily nested patterns, wildcards, multiple branches
  - as-patterns, irrefutable patterns, guards, etc. would be nice too
- Must agree with standard denotational semantics for patterns
  - Should use a maybe monad for semantics
- After pretty printing, it must look like a case statement!
  - Should use Isabelle's macro mechanism for parsing/printing
  - This means we can't generate fresh variable names

Semantics of Case Expressions in HOLCF

- Consider the following expression, where `x::'a` and the whole expression has type `'b`

```
Case x of pat1 => rhs1 | pat2 => rhs2 | ...
```

  - Each branch has type `'a -> 'b maybe`
  - Branches are combined using fatbar and run operators

```
consts
  fatbar :: ('a -> 'b maybe) -> ('a -> 'b maybe)
              -> ('a -> 'b maybe)
  run :: 'a maybe -> 'a
translations
  "Case x of ms" == "run (ms x)"
  "m | ms" == "fatbar m ms"
```

Semantics of Case Branches in HOLCF

- Consider the case branch `pat => rhs`, which has type `'a -> 'b maybe`
  - Let `'c` be the type of a tuple containing all values bound by `pat`
  - `pat` has type `'a -> 'c maybe`
  - `rhs` has type `'b -> 'c`
  - They are combined using the `branch` operator

```
branch :: ('a -> 'c maybe) -> ('c -> 'b)
              -> ('a -> 'b maybe)
```

Pattern Combinators

```
constdefs
  wildP :: 'a -> unit maybe
  wildP == LAM x. return ()

  varP :: 'a -> 'a maybe
  varP == LAM x. return x

  cpairP :: ('a -> 'c maybe) => ('b -> 'd maybe)
            -> (('a * b) -> ('c * 'd) maybe)
  cpairP p1 p2 ==
    LAM <x,y>. do a <- p1 x; b <- p2 y; return <a,b>

  lazyP :: ('a -> 'b maybe) => ('a -> 'b maybe)
  lazyP p == LAM x. return (run (p x))
```

More Pattern Combinators

Define pattern combinators for other data constructors using cpairP

```
domain 'a list = Nil | Cons (lazy 'a) (lazy 'a list)

constdefs
  NilP :: 'a list -> unit maybe
  NilP == LAM xs.
    case xs of Nil => return () | Cons x xs => fail

  ConsP :: ('a -> 'b maybe) => ('a list -> 'c maybe)
           => ('a list -> ('b * 'c) maybe)
  ConsP p1 p2 == LAM xs.
    case xs of Nil => fail | Cons x xs => cpairP p1 p2 <x,xs>
```

Simplification of Case Expressions

Rules for simplifying with `fatbar`:

```
m x = ⊥        ==> (fatbar m ms) x = ⊥
m x = fail      ==> (fatbar m ms) x = ms x
m x = return y ==> (fatbar m ms) x = return y
```

Rules for simplifying with `cpairP`:

```
branch p1 r x = ⊥
  ==> branch (cpairP p1 p2) (csplit r) <x,y> = ⊥
branch p1 r x = fail
  ==> branch (cpairP p1 p2) (csplit r) <x,y> = fail
branch p1 r x = return s
  ==> branch (cpairP p1 p2) (csplit r) <x,y> = branch p2 s y
```

Syntax of Case Expressions in HOLCF

- In Isabelle, all variable binding is done with lambda abstractions
- Other variable binding syntax is translated to lambdas
  - One abstraction per bound variable

```
translations
  "ALL x. P" == "ALL (λx. P)"
  "λ(x,y). b" == "split (λx. λy. b)"
```

- Challenge: Nested patterns may bind any number of variables

```
"C x (C y z) => rhs" == "... (LAM <x, <y, z>>. rhs)"
```

Pretty Printing for Case Expressions

```
run ((branch (ConsP (cpairP varP wildP) varP)
             (LAM <<x,()>,ys>. rhs)) xs)
Case xs of (branch (ConsP (cpairP varP wildP) varP)
                   (LAM <<x,()>,ys>. rhs))
Case xs of (_PAT (ConsP (cpairP varP wildP) varP)
                 ((x,()),ys)) => rhs)
Case xs of (Cons (_PAT (cpairP varP wildP) (x,()))
                 (_PAT varP ys) => rhs)
Case xs of (Cons <_PAT varP x, _PAT wildP ()>
                 (_PAT varP ys) => rhs)
Case xs of Cons <x,_> ys => rhs
```

Parsing for Case Expressions

```
Case xs of Cons <x,_> ys => rhs
run ((Cons <x,_> ys => rhs) xs)
run ((branch (_PAT (Cons <x,_> ys))
             (LAM _VAR (Cons <x,_> ys). rhs)) xs)
run ((branch (ConsP (_PAT <x,_>) (_PAT ys))
             (LAM <_VAR <x,_>, _VAR ys>. rhs)) xs)
run ((branch (ConsP (cpairP (_PAT x) (_PAT _)) (_PAT ys))
             (LAM <<_VAR x, _VAR _>, _VAR ys>. rhs)) xs)
run ((branch (ConsP (cpairP varP wildP) varP)
             (LAM <<x, ()>, ys>. rhs)) xs)
```

More Haskell-like Features

Haskell-style expression syntax:

- Patterns in Lambda abstractions
- Letrec syntax
- "Haskell brackets"

Future work:

- Translating type classes