Haskell to Isabelle/HOLCF

Brian Huffman

Programatica Talk
October 28, 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

- Currently disallowed features:
  - full case expressions with nested patterns
  - local function 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
  "%(x,y,zs).b" == "split(%x (y,zs).b)"
  "%(x,y).b" == "split(%x y. 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 matches" == "run (matches x)"
  "match | matches" == "fatbar match matches"
```

Semantics of Case Branches 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 (%x. P)"
  "%(x,y). b" == "split (%x y. b)"
```

- Challenge: Nested patterns may bind any number of variables

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

Pattern Matching Combinators

```
constdefs
  wildP :: 'b -> 'a -> 'b maybe
  wildP == LAM r a. return r
  varP :: ('a -> 'b) -> 'a -> 'b maybe
  varP == LAM r a. return (r a)
  asP :: ... => ('a -> 'b) -> 'a -> 'c maybe
  asP P == LAM r a. P (r a) a

translations
  "_ => r" == "wildP r"
  "x => r" == "varP (LAM x. r)"
  "_as1 x (P r)" == "asP P (LAM x. r)"
  "x as p => r" == "_as1 x (p => r)"
```

Pattern Matching for Pairs

Intermediate constants are used to apply pattern arguments in order

```
defs
  cpairP P1 P2 == LAM r <x,y>. do r' <- P1 r x; P2 r' y

translations
  "_cpair1 (P1 r) P2" == "(cpairP P1 P2) r"
  "_cpair2 p1 (P2 r)" == "_cpair1 (p1 => r) P2"
  "<p1, p2> => r"     == "_cpair2 p1 (p2 => r)"
```

Matching combinators for other constructors are defined in terms of `cpairP`

Simplification of Case Expressions

Rules for simplifying with `fatbar`:

```
P x = ⊥        ==> run ((fatbar P Ps) x) = ⊥
P x = fail     ==> run ((fatbar P Ps) x) = run (Ps x)
P x = return y ==> run ((fatbar P Ps) x) = y
```

Rules for simplifying with `cpairP`:

```
P1 r x = ⊥         ==> cpairP P1 P2 r <x,y> = ⊥
P1 r x = fail      ==> cpairP P1 P2 r <x,y> = fail
P1 r x = return r' ==> cpairP P1 P2 r <x,y> = P2 r' y
```

Matching Combinators for Other Constructors

- Pattern match combinator for `Cons` is defined in terms of `cpairP`

```
ConsP P1 P2 r Nil = fail
ConsP P1 P2 r (Cons x xs) = cpairP P1 P2 r <x,xs>
```

- Using these as rewrite rules, the simplifier can show:

```
(Case Cons x xs of Cons p1 p2 => rhs)
  = (Case <x,xs> of <p1,p2> => rhs)
```

- This way we can reuse the simp rules for `cpairP` (less theorems to generate)

- But we do need to generate syntax macros for `ConsP`

Translating P-logic

- Syntax of P-logic predicates (from "The Logic of Demand in Haskell")

```
data Pr = Univ | UnDef | ConPred Name [Pr] | Strong Pr
        | PredVar Name | PArrow Pr Pr | Pneg Pr
```

- Full P-logic actually has more constructors than these

- Predicates translate to sets in Isabelle

- Each constructor translates to an Isabelle constant

Definitions of P-logic Operations

```
strong :: 'a set => 'a set
strong A == A - {⊥}

arrow :: 'a set => 'b set => ('a -> 'b) set
arrow A B == {f. ALL x:A. f x : B}

conpred1 :: ('a -> 'b) => 'a set => 'b set
conpred1 f A == {x. EX a:A. x = f a}

conpred2 :: ('a -> 'b -> 'c) =>
              'a set => 'b set => 'c set
conpred2 f A B == {x. EX a:A. EX b:B. x = f a b}
```

Weakening Operator

- To make the semantics agree with the intended semantics for P-logic, we also need a weakening operator:

```
weak :: 'a set => 'a set
weak A == insert ⊥ A
```

- This operator is implicit in P-logic, but it is present everywhere that does not have the strengthening operator!

- Proposal: Make the weakening operator *explicit*, and make it *optional*
  - Validity of inference rules would be much simpler to understand
  - Inference rules could be written that require predicates to be weak