Haxell
adding regular types to Haskell

Matthew Fuchs

(Satnam Singh)

Microsoft

- Original goal – an Xduce style type system for a message passing language
- New goal – full regular type integration into Haskell
  - Smooth integration of regular and algebraic types
  - Support for infinite hedges

Other Work

- Similar to Cduce in pattern matching
  - Does not support negative patterns
- Very similar to XHaskell except
  - intend to integrate type checking into the compiler, rather than continue with generating types
  - intend to support parametric polymorphism
  - will require less type annotations
    - the compiler “knows” which subsumption checks need to be made

Notation

- <foo>True</foo><bar>10</bar><bar>20</bar><baz>“some text”</baz>
- :foo[True] :bar[10] :bar[20] :baz[“some text”]
- Type:
  - (| :foo[Bool], :bar[Integer]*, :baz[String] |)
  - types nailed down quite specifically
- Change all the element names from foo to oof, bar to rab, and baz to zab:
  - flipName:: (| :foo[Bool], :bar[Integer]*, :baz[String] |) -> (| :oof[Bool], :rab[Integer]*, :zab[String] |)
  - even though contents of foo, bar, and baz are irrelevant

Type variables

- Haskell type variables:
  - (| :foo[t1], :bar[t2]*, :baz[t3] |)
- flipName:: (| :foo[t1], :bar[t2]*, :baz[t3] |) -> (| :oof[t1], :rab[t2]*, :zab[t3] |)
- Add elements:
  - addem (|:foo[_], :bar[x]*, :baz[_]|) = fold (+) x
- Types:
  - addem:: (| :foo[t1], :bar[t2]*, :baz[t3] |)-> Integer
  - addem::(Num t2) => (| :foo[t1], :bar[t2]*, :baz[t3] |)-> t2

New types

- Foo t1 t2 t3 = (| :foo[t1], :bar[t2]*, :baz[t3] |)
- Bar = Foo String Integer Bool

Weak Matching

- resolve ambiguity through weak matching:
  - (| (:foo[], t1) | (t2, :bar[]) |)
  - t2 could match :foo and t1 could match :bar
  - weak matching, we exclude t2 from starting with :foo and thereby resolve the ambiguity

Pattern Matching

- addem (|:foo[_], :bar[x@Integer]*, :baz[_]|) = fold (+) x
- addem (|:foo[_], :bar[x@Float]*, :baz[_]|) = fold (*) x
- addem (|:foo[_], :bar[x@Double]*, :baz[_]|) = fold (/) x

Current state

- Extended syntax (hacked Haskell grammar)
- Translation to “standard” Haskell (may require some apparently obscure declarations)
- Subsumption checks are performed once (if “–O”) at runtime and then resolve to True
- Pattern matching closer to Cduce than Xduce
- Marshal/Unmarshal between Haskell’s types and regular types is possible as defined by user.

Implementation

- A `newtype` for every regular type either
  - Defined by programmer, or
  - Generated from a pattern in a match
- Each pattern also generates a tuple type corresponding to the bound variables
- Also instance of class XMark providing info for
  - Validation
  - Casting to/from the type
  - Pattern matching – pattern match returns Either tuple error-message
- Finally, a structure containing all regex's is generated for validating and pattern matching.

Example regex type

```
regex Envelope =
  (|:envelope[:headers[Header**]??,
    :body[Any**]]|)
```

becomes

```
newtype Envelope = Envelope [Tree]
Instance XMark Envelope () where
    -- “type” a hedge as an Envelope
    create x = Envelope x
    -- remove type so x can be cast to another type
    decreate (Envelope x) = x
    -- type name to be passed to validator
    value x = “Envelope”
```

Pattern example

```
f (|:integers[int@Integer]**
    :strings[str@String]**|) = ….
```

becomes

```
newtype Gensym0 = Gensym0 [Tree]
type Gensym1 = ([Integer], [String])
instance XMark Gensym0 Gensym1 where
   … as above …
   -- pattern variable names
   varList x = [“int”, “str”]
   -- convert results of pattern match to a Gensym1 tuple
   toTuple _ (Right [int, str]) =
         Just (create int, create str)
   toTuple _ (Left errormessage) = error errormessage
```

Implementation, cont.

- Instances of Cast where trees must be cast from one type to another
  - given `foo::Foo -> Bar` and `bar::Bar`, `(foo bar)` requires Bar <: Foo check
  - given `instance Cast Bar Foo`,
    - `foo (cast bar)` does the right thing
    - One runtime check (if (Bar <: Foo) then …) becomes (if True then …) if compiled with –O
- but these instances must currently be added manually
  - programmer changes `(foo bar)` to `(foo (cast bar))`
  - compiler will list all the instances to be inserted.
  - next step is to automate this

Implementation, cont.

- Serialize/Deserialize
  - `class Castor regtype algebraic section`
    - Given `deserialize::section->algebraic` then for any regtype, given `cast::regtype-> section` there is `deserialize::regtype-> algebraic`
  - `class Xmlable algebraic regtype`
- Much of this is very similar to patterns in “Scrap Your Boilerplate”
  - in particular, our “cast” is a slight generalization of the one in that paper.

Future Goals

- Compile time type checking inside GHC
  - remove most run time checks and generated code
- Default (de)serialization for algebraic types with user override
- Parametric polymorphism and type inference supporting infinite hedges

laziness and (non)determinism

Suppose we call a function with an infinite hedge:

```
f [([ :int[x] ]) | x <- [1..]]
```

Suppose we have patterns:

- `f(|:int[a@Int]?,(:int[b@Int],:int[c@Int])*|)=…`

  no assignment until finished and will diverge while matching

- `f(|:int[a@Int]?|)= …`

  `f(|(:int[b@Int],:int[c@Int])+|)=…`

  `f(|:int[a@Int],(:int[b@Int],:int[c@Int])+|)=…`

  each deterministic, but cannot be distinguished in bounded time, so it will diverge

- `f(|(:int[b@Int],:int[c@Int])*,:int[a@Int]?|)=…`
  matches in bounded time, allows processing of b and c, but accessing a will diverge

- Preference for the last alternative – matching and assignment should happen in bounded time, as in ordinary Haskell.

What's a hedge?

- Hedge

  ```
  ([:foo[10 12 13] :bar[:foo["abcde", :bar[]]]]) =>
  <foo>10 12 13</foo> <bar><foo>abcde</foo><bar/></bar>
  ```

- Regular expression type

  ```
  regex Foo = (|:foo[Integer** | String], :bar[Foo??]|)
  ```

  `[namespace]:`name for tag

  "|" separates choice, "," separates sequence, will add "&" for unordered

- Pattern (notice the variable bindings)

  ```
  f(|:foo[(intList@Integer)** | astr@String], _|) =
    (strval,intval)
   where
    strval= if ((length astr)>0)
                then (head astr) else "",
    intval = foldl (+) 0 intList)
  ```