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>
- 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

\[
f \left( \mid \text{integers}[\text{int}\@\text{Integer}] \mid \text{strings}[\text{str}\@\text{String}] \mid \right) = \ldots
\]

becomes

\[
\text{newtype Gensym0} = \text{Gensym0} \ [\text{Tree}]
\]
\[
\text{type Gensym1} = ([\text{Integer}], [\text{String}])
\]
\[
\text{instance XMark Gensym0 Gensym1 where}
\]
\[
\ldots \text{as above} \ldots
\]
\[
\quad \text{-- pattern variable names}
\]
\[
\text{varList x} = ["\text{int}", "\text{str}"]
\]
\[
\quad \text{-- convert results of pattern match to a Gensym1 tuple}
\]
\[
\text{toTuple} \_ \ (\text{Right} [\text{int}, \text{str}]) =
\]
\[
\quad \text{Just (create int, create str)}
\]
\[
\text{toTuple} \_ \ (\text{Left errormessage}) = \text{error errormessage}
\]
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 Xmlableable 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:

\[ f \left( \left( \left[ :\text{int}[x] \right] \right) \mid x \leftarrow [1..] \right) \]

Suppose we have patterns:

- \( f(|:\text{int}[a@\text{Int}]?,(:\text{int}[b@\text{Int}], :\text{int}[c@\text{Int}])*|) = \ldots \)
  - no assignment until finished and will diverge while matching
- \( f(|:\text{int}[a@\text{Int}]?|) = \ldots \)
- \( f(|(:\text{int}[b@\text{Int}], :\text{int}[c@\text{Int}])+|) = \ldots \)
- \( f(|:\text{int}[a@\text{Int}], (:\text{int}[b@\text{Int}], :\text{int}[c@\text{Int}])+|) = \ldots \)
  - each deterministic, but cannot be distinguished in bounded time, so it will diverge
- \( f(|(:\text{int}[b@\text{Int}], :\text{int}[c@\text{Int}])*,:\text{int}[a@\text{Int}]?|) = \ldots \)
  - 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
  \(<foo>10 12 13</foo> <bar><foo>abcde</foo><bar/></bar>\)

• Regular expression type
  \(\text{regex } \text{Foo} = (|:\text{foo}[\text{Integer}\ddagger | \text{String}], :\text{bar}[\text{Foo}\ddagger]|)\)
  \[[\text{namespace}]\text{:name}\text{ for tag}\]
  “|” separates choice, “,” separates sequence, will add “&” for unordered

• Pattern (notice the variable bindings)
  \(f(|:\text{foo}[\text{(intList@Integer)}\ddagger | \text{astr@String}], _|) =\)
  \((\text{strval},\text{intval})\)
  \where\)
    \text{strval}= \text{if } ((\text{length }\text{astr})>0)
    \text{then } (\text{head }\text{astr}) \text{ else ““},
    \text{intval }= \text{foldl (} (+) 0 \text{ intList})