Formalism for Datatype Extensions in Haskell

Tim Zakian

July 15, 2015

1 Language Definition

- Why does this not create open datatypes? How does it differ and how does it get around the problems found in open datatypes?

2 Typing Rules

Symbol Classes

$$
\begin{array}{lcl}
a, b, c, co & \to & \langle \text{type variable} \rangle \\
x, f & \to & \langle \text{term variable} \rangle \\
C & \to & \langle \text{coercion constant} \rangle \\
T & \to & \langle \text{value type constructor} \rangle \\
S_n & \to & \langle n - \text{ary type function} \rangle \\
K & \to & \langle \text{data constructor} \rangle
\end{array}
$$

Declarations

$$
\begin{array}{lcl}
pgm & \to & \overline{decl}; e \\
decl & \to & \textbf{data}\ T\ :\ \overline{\kappa} \to \star\ \textbf{where} \\
 & & \quad \overline{K : \forall\ \overline{a : \kappa}.\ \forall\ \overline{b : \iota}.\overline{\sigma} \to T\ \overline{a}} \\
 & | & \textbf{type}\ S_n\ :\ \overline{\kappa}^n \to \iota \\
 & | & \textbf{axiom}\ C\ :\ \sigma_1 \sim \sigma_2
\end{array}
$$

Sorts and Kinds

$$
\begin{array}{lcll}
\delta & \to & TY \mid CO & \text{Sorts} \\
\kappa, \iota & \to & \star \mid \kappa_1 \to \kappa_2 \mid \sigma_1 \sim \sigma_2 & \text{Kinds}
\end{array}
$$

Types and Coercions

$$
\begin{array}{lcll}
d & \to & a \mid T & \text{Atom of sort TY} \\
g & \to & c \mid C & \text{Atom of sort CO} \\
\varphi, \rho, \sigma, \tau, \nu, \gamma & \to & a \mid C \mid T \mid \varphi_1\ \varphi_2 \mid S_n\ \overline{\varphi}^n \mid \forall a : \kappa.\varphi & \\
 & | & \text{sym}\ \gamma \mid \gamma_1 \circ \gamma_2 \mid \gamma@\varphi \mid \text{left}\ \gamma \mid \text{right}\ \gamma & \\
 & | & \gamma \sim \gamma \mid \text{rightc}\ \gamma \mid \text{leftc}\ \gamma \mid \gamma \blacktriangleright \gamma &
\end{array}
$$

Syntactic sugar

Types $\kappa \Rightarrow \sigma\ \equiv\ \forall\_ : \kappa.\sigma$

Terms

$$
\begin{array}{lcll}
u & \to & x \mid K & \text{Variables and data consructors} \\
e & \to & u & \text{Term atoms} \\
 & | & \Lambda a : \kappa.e \mid e\varphi & \text{Type abstractions/application} \\
 & | & \lambda x : \sigma.e \mid e_1\ e_2 & \text{Term abstraction/application} \\
 & | & \textbf{let}\ x : \sigma = e_1\ \textbf{in}\ e_2 & \\
 & | & \textbf{case}\ e_1\ \textbf{of}\ \overline{p \to e_2} & \\
 & | & e \blacktriangleright \gamma & \text{Cast} \\
 & & & \\
p & \to & K\ \overline{b : \kappa}\ \overline{x : \sigma} & \text{Pattern}
\end{array}
$$

Environments

$\Gamma \quad \to \quad \epsilon \mid \Gamma, u : \sigma \mid \Gamma, d : \kappa \mid \Gamma, g : \kappa \mid \Gamma, S_n : \kappa$

A top-level environment binds only type constructors,
T, S_n, data constructors K, and coercion constants C.

Figure 1: The core language for extensible data types