Formalism for Datatype Extensions in Haskell

Tim Zakian

July 26, 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?
- Currently only have core $F_C(X)$ as presented in [1]. Need to extend this with the extendable data type section (i.e., the hard stuff...).

Symbol Classes

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

Sorts and Kinds

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

Syntactic sugar

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

Types and Coercions

$$\begin{array}{llll}
d & \to & a \mid T & \text{Atom of sort TY} \\
g & \to & c \mid C & \text{Atom of sort CO} \\
\varphi, \rho, \sigma, & & & \\
\tau, \nu, \gamma & \to & a \mid C \mid T \mid \varphi_1\ \varphi_2 \mid S_n\ \overline{\varphi}^n \mid \forall a : \kappa.\varphi & \\
 & \mid & \mathrm{sym}\ \gamma \mid \gamma_1 \circ \gamma_2 \mid \gamma @ \varphi \mid \mathrm{left}\ \gamma \mid \mathrm{right}\ \gamma & \\
 & \mid & \gamma \sim \gamma \mid \mathrm{rightc}\ \gamma \mid \mathrm{leftc}\ \gamma \mid \gamma \blacktriangleright \gamma & \\
 & \mid & \zeta_1 \underset{[T_1, T_2]}{\sim} \zeta_2 & \\
\zeta & \to & (K, \sigma) \mid \sigma &
\end{array}$$

Environments

$\Gamma \quad \to \quad \epsilon \mid \Gamma, u : \sigma \mid \Gamma, d : \kappa \mid \Gamma, g : \kappa \mid \Gamma, S_n : \kappa \mid \Gamma, (T : \overline{\zeta})$

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

Declarations

$$\begin{array}{lll}
pgm & \to & \overline{cdecl}; \overline{decl}; e \\
cdecl & \to & \textbf{data extendable}\ T\ :\ \overline{\kappa} \to \star\ \textbf{where} \\
 & & \quad \overline{\mathtt{K} : \forall\, \overline{a : \kappa}.\ \forall\, \overline{b : \iota}.\overline{\sigma} \to T\ \overline{a}} \\
 & \mid & \textbf{data}\ T\ :\ \overline{\kappa} \to \star\ \textbf{where} \\
 & & \quad \overline{\mathtt{K} : \forall\, \overline{a : \kappa}.\ \forall\, \overline{b : \iota}.\overline{\sigma} \to T\ \overline{a}} \\
 & & \textbf{extends}(\overline{T}) \\
 & \mid & \textbf{data extendable}\ T\ :\ \overline{\kappa} \to \star\ \textbf{where} \\
 & & \quad \overline{\mathtt{K} : \forall\, \overline{a : \kappa}.\ \forall\, \overline{b : \iota}.\overline{\sigma} \to T\ \overline{a}} \\
 & & \textbf{extends}(\overline{T}) \\
decl & \to & \textbf{data}\ T\ :\ \overline{\kappa} \to \star\ \textbf{where} \\
 & & \quad \overline{\mathtt{K} : \forall\, \overline{a : \kappa}.\ \forall\, \overline{b : \iota}.\overline{\sigma} \to T\ \overline{a}} \\
 & \mid & \textbf{type}\ S_n\ :\ \overline{\kappa}^n \to \iota \\
 & \mid & \textbf{axiom}\ C\ :\ \sigma_1 \sim \sigma_2
\end{array}$$

Terms

$$\begin{array}{llll}
u & \to & x \mid K & \text{Variables and data consructors} \\
e & \to & u & \text{Term atoms} \\
 & \mid & \Lambda a : \kappa.e \mid e\varphi & \text{Type abstractions/application} \\
 & \mid & \lambda x : \sigma.e \mid e_1\ e_2 & \text{Term abstraction/application} \\
 & \mid & \textbf{let}\ x : \sigma = e_1\ \textbf{in}\ e_2 & \\
 & \mid & \textbf{case}\ e_1\ \textbf{of}\ \overline{p \to e_2} & \\
 & \mid & e \blacktriangleright \gamma \mid e \underset{[T_1, T_2]}{\triangleright} \gamma & \text{Cast} \\
 & & & \\
p & \to & K\ \overline{b : \kappa}\ \overline{x : \sigma} & \text{Pattern}
\end{array}$$

Figure 1: The core language for extensible data types – $F_C(X)$

2 Typing Rules

2.1 Extended typing rules

TODO:

- **Explain what cast extrusion is and why it's bad**
- Need to formalize "Get all the data constructors for this type constructor" in this system since we will need it for ExtData in our environment extension rules.
- We need the extra equivalences since we need to prevent casts between differenet constructors that have the same type.
- **Important:** Need to ensure that the type constructors that we are extending are the same kind as the one that extends it.

We now extend the core typing rules for System F_C with a way to cast between extended and core data constructors, while at the same time preventing "cast extrusion" on the constructors and corresponding types.

Notation: Throughout this section we use the notation K_T to represent a given constructor K under a specific type constructor context e.g., K_T would be that same constructor as $K_{T'}$ except that K_T produces a type T and $K_{T'}$ produces a type T'.

Definition We say that $\sigma_1 \underset{[T_1,T_2]}{\equiv} \sigma_2$ if:

$$\sigma_1 = \forall \overline{a : \kappa} \forall \overline{b : \iota}. \overline{\sigma} \to T_1\ \overline{a}$$

and

$$\sigma_2 = \forall \overline{a : \kappa} \forall \overline{b : \iota}. \overline{\sigma} \to T_2\ \overline{a}$$

i.e., that they are they same type modulo substitution of the type constructor.

Definition: We define $\zeta[\![T/T']\!]$ to be the substitution of the type constructor application the underlying type σ.

Definition: We say that T extends T' and that $T > T'$ if T is a type constructor derived using a **extends** form that includes T'.

Theorem 1 (Transitivity of $(>)$)**.** *Let $T_1 > T_2$ and $T_2 > T_1$. Then $T_1 > T_3$.*

Proof. Let K_3, K_2, K_1 be the set of constructors for T_3, T_2, T_1 respectively. Then we have that $K_3 \subset K_2 \subset K_1$. Since $T_2 > T_3$ we have that $\forall K \in K_3$ there exists a cast γ such that

$$K_3 \ni (K : \sigma) \underset{[T_3,T_2]}{\triangleright} \gamma : (K : \sigma') \in K_2 \tag{1}$$

Similarly, since $T_1 > T_2$. We have that $\forall K \in K_2$ that there exists a γ' such that

$$K_2 \ni (K : \sigma') \underset{[T_2,T_1]}{\triangleright} \gamma' : (K : \sigma'') \in K_1 \tag{2}$$

We therefore have by transitivity of $\underset{[T,T']}{\sim}$ that $\forall K \in K_3$ that

$$K_3 \ni (K : \sigma) \underset{[T_3,T_1]}{\triangleright} \gamma \circ \gamma' : (K : \sigma'') \in K_1 \tag{3}$$

and hence, we have that $T_1 > T_3$ □

Theorem 2 ($[\![T/T']\!]$ preserves well-typedness)**.** *Let $\Gamma \vdash_e K_T : \sigma$ and $\Gamma \vdash_e K_{T'} : \sigma'$. Then we have that $\sigma' = \sigma[\![T/T']\!]$ – i.e., that if $\Gamma \vdash_e K_T : \sigma$, then $\Gamma \vdash_e K_{T'} : \sigma[\![T/T']\!]$.*

Proof. WRITE ME □

Theorem 3 (Extension casts preserve well-typedness)**.** *Proof.* WRITE ME □

$$\boxed{\Gamma \vdash_{\mathrm{TY}} \sigma : \kappa}$$

$$\text{TyVar}\,\frac{(d : \kappa) \in \Gamma \qquad \Gamma \vdash_{\mathrm{k}} \kappa : TY}{\Gamma \vdash_{\mathrm{TY}} d : \kappa} \qquad \text{TyApp}\,\frac{\Gamma \vdash_{\mathrm{TY}} \sigma_1 : \kappa_1 \to \kappa_2 \qquad \Gamma \vdash_{\mathrm{TY}} \sigma_2 : \kappa_1}{\Gamma \vdash_{\mathrm{TY}} \sigma_1\ \sigma_2 : \kappa_2}$$

$$\text{TySCon}\,\frac{(S_n : \overline{\kappa}^n \to \iota) \in \Gamma \qquad \Gamma \vdash_{\mathrm{TY}} \overline{\sigma : \kappa}^n}{\Gamma \vdash_{\mathrm{TY}} S_n \overline{\sigma}^n : \iota} \qquad \text{TyAll}\,\frac{\Gamma, a : \kappa \vdash_{\mathrm{TY}} \sigma : \star \qquad \Gamma \vdash_{\mathrm{k}} \kappa : \delta \qquad a \notin fv(\Gamma)}{\Gamma \vdash_{\mathrm{TY}} \forall a : \kappa . \sigma : \star}$$

$$\boxed{\Gamma \vdash_{\mathrm{CO}} \gamma : \sigma \sim \tau}$$

$$\text{CoRefl}\,\frac{(a : \kappa) \in \Gamma \qquad \Gamma \vdash_{\mathrm{k}} \kappa : TY}{\Gamma \vdash_{\mathrm{CO}} a : a \sim a} \qquad \text{CoVar}\,\frac{(g : \sigma \sim \tau) \in \Gamma}{\Gamma \vdash_{\mathrm{CO}} g : \sigma \sim \tau} \qquad \text{CoAllT}\,\frac{\begin{array}{c}\Gamma, a : \kappa \vdash_{\mathrm{CO}} \gamma : \sigma \sim \tau \\ \Gamma \vdash_{\mathrm{k}} \kappa : TY \qquad a \notin fv(\Gamma)\end{array}}{\Gamma \vdash_{\mathrm{CO}} \forall a : \kappa . \gamma : \forall a : \kappa . \sigma \sim \forall a : \kappa . \tau}$$

$$\text{CoInstT}\,\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{CO}} \gamma : \forall a . \kappa . \sigma \sim \forall b : \kappa . \tau \\ \Gamma \vdash_{\mathrm{TY}} v : \kappa\end{array}}{\Gamma \vdash_{\mathrm{CO}} \gamma @ v : [v/a]\sigma \sim [v/b]\tau} \qquad \text{SComp}\,\frac{\Gamma \vdash_{\mathrm{CO}} \overline{\gamma : \sigma \sim \tau}^n \qquad \Gamma \vdash_{\mathrm{TY}} S_n \overline{\sigma}^n : \kappa}{\Gamma \vdash_{\mathrm{CO}} S_n \overline{\gamma}^n : S_n \overline{\sigma}^n \sim S_n \overline{\tau}^n} \qquad \text{Sym}\,\frac{\Gamma \vdash_{\mathrm{CO}} \gamma : \sigma \sim \tau}{\Gamma \vdash_{\mathrm{CO}} \gamma : \tau \sim \sigma}$$

$$\text{Trans}\,\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{CO}} \gamma_1 : \sigma_1 \sim \sigma_2 \\ \Gamma \vdash_{\mathrm{CO}} \gamma_2 : \sigma_2 \sim \sigma_3\end{array}}{\Gamma \vdash_{\mathrm{CO}} \gamma_1 \circ \gamma_2 : \sigma_1 \sim \sigma_3} \qquad \text{Comp}\,\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{CO}} \gamma_1 : \sigma_1 \sim \tau_1 \qquad \Gamma \vdash_{\mathrm{CO}} \gamma_2 : \sigma_2 \sim \tau_2 \\ \Gamma \vdash_{\mathrm{TY}} \sigma_1\ \sigma_2 : \kappa\end{array}}{\Gamma \vdash_{\mathrm{CO}} \gamma_1\ \gamma_2 : \sigma_1\ \sigma_2 \sim \tau_1\ \tau_2}$$

$$\text{Left}\,\frac{\Gamma \vdash_{\mathrm{CO}} \gamma : \sigma_1\ \sigma_2 \sim \tau_1\ \tau_2}{\Gamma \vdash_{\mathrm{CO}} \mathrm{left}\ \gamma : \sigma_1 \sim \tau_1} \qquad \text{Right}\,\frac{\Gamma \vdash_{\mathrm{CO}} \gamma : \sigma_1\ \sigma_2 \sim \tau_1\ \tau_2}{\Gamma \vdash_{\mathrm{CO}} \mathrm{right}\ \gamma : \sigma_1 \sim \tau_1}$$

$$\text{CompC}\,\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{CO}} \gamma : \kappa_1 \sim \kappa_2 \qquad \Gamma \vdash_{\mathrm{CO}} \gamma' : \sigma_1 \sim \sigma_2 \\ \Gamma \vdash_{\mathrm{k}} \kappa_1 : CO\end{array}}{\Gamma \vdash_{\mathrm{CO}} \gamma \Rightarrow \gamma' : (\kappa_1 \Rightarrow \sigma_1) \sim (\kappa_2 \Rightarrow \sigma_2)} \qquad \text{LeftC}\,\frac{\Gamma \vdash_{\mathrm{CO}} \gamma : \kappa_1 \Rightarrow \sigma_1 \sim \kappa_2 \Rightarrow \sigma_2}{\Gamma \vdash_{\mathrm{CO}} \mathrm{leftc}\ \gamma : \kappa_1 \sim \kappa_2}$$

$$\text{RightC}\,\frac{\Gamma \vdash_{\mathrm{CO}} \gamma : \kappa_1 \Rightarrow \sigma_1 \sim \kappa_2 \Rightarrow \sigma_2}{\Gamma \vdash_{\mathrm{CO}} \mathrm{rightc}\ \gamma : \sigma_1 \sim \sigma_2} \qquad (\sim)\,\frac{\Gamma \vdash_{\mathrm{CO}} \gamma_1 : \sigma_1 \sim \tau_1 \qquad \Gamma \vdash_{\mathrm{CO}} \gamma_2 : \sigma_2 \sim \tau_2}{\Gamma \vdash_{\mathrm{CO}} \gamma_1 \sim \gamma_2 : (\sigma_1 \sim \sigma_2) \sim (\tau_1 \sim \tau_2)}$$

$$\text{CastC}\,\frac{\Gamma \vdash_{\mathrm{CO}} \gamma_1 : \kappa \qquad \Gamma \vdash_{\mathrm{CO}} \gamma_2 \kappa \sim \kappa'}{\Gamma \vdash_{\mathrm{CO}} \gamma_1 \blacktriangleright \gamma_2}$$

$$\boxed{\Gamma \vdash_{\mathrm{e}} e : \zeta}$$

$$\text{Var}\,\frac{(u : \sigma) \in \Gamma}{\Gamma \vdash_{\mathrm{e}} u : \sigma} \qquad \text{Case}\,\frac{\Gamma \vdash_{\mathrm{e}} e : \sigma \qquad \overline{\Gamma \vdash_{\mathrm{p}} p \to e : \sigma \to \tau}}{\Gamma \vdash_{\mathrm{e}} \mathbf{case}\ e\ \mathbf{of}\ \overline{p \to e} : \tau} \qquad \text{Let}\,\frac{\Gamma \vdash_{\mathrm{e}} e_1 : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash_{\mathrm{e}} e_2 : \sigma_2}{\Gamma \vdash_{\mathrm{e}} \mathbf{let}\ x : \sigma_1 = e_1\ \mathbf{in}\ e_2 : \sigma_2}$$

$$\text{Cast}\,\frac{\Gamma \vdash_{\mathrm{e}} e : \sigma \qquad \Gamma \vdash_{\mathrm{CO}} \gamma : \sigma \sim \tau}{\Gamma \vdash_{\mathrm{e}} e \blacktriangleright \gamma : \tau} \qquad \text{Abs}\,\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{TY}} \sigma_x : \star \\ \Gamma, x : \sigma_x \vdash_{\mathrm{e}} e : \sigma\end{array}}{\Gamma \vdash_{\mathrm{e}} \lambda x : \sigma_x . e : \sigma_x \to \sigma} \qquad \text{App}\,\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{e}} e_1 : \sigma_2 \to \sigma_1 \\ \Gamma \vdash_{\mathrm{e}} e2 : \sigma_2\end{array}}{\Gamma \vdash_{\mathrm{e}} e_1\ e_2 : \sigma_1}$$

$$\text{AbsT}\,\frac{\Gamma a : \kappa \vdash_{\mathrm{e}} e : \sigma \qquad \Gamma \vdash_{\mathrm{k}} \kappa : \delta \qquad a \notin fv(\Gamma)}{\Gamma \vdash_{\mathrm{e}} \Lambda a : \kappa . e : \forall a : \kappa . \sigma} \qquad \text{AppT}\,\frac{\Gamma \vdash_{\mathrm{e}} e : \forall a : \kappa . \sigma \qquad \Gamma \vdash_{\mathrm{k}} \kappa : \delta \qquad \Gamma \vdash_{\delta} \varphi : \kappa}{\Gamma \vdash_{\mathrm{e}} e\ \varphi : \sigma[\varphi/a]}$$

$$\boxed{\Gamma \vdash_{\mathrm{p}} p \to e : \sigma \to \tau}$$

$$\text{Alt}\,\frac{K : \forall \overline{a : \kappa} . \forall \overline{b : \iota} . \overline{\sigma} \to T\ \overline{a} \in \Gamma \qquad \theta = [\overline{v/a}] \qquad \Gamma, \overline{b : \theta(\iota), \overline{x : \theta(\sigma)}} \vdash_{\mathrm{e}} e : \tau}{\Gamma \vdash_{\mathrm{p}} K\ \overline{b : \theta(\iota)}\ \overline{x : \theta(\sigma)} \to e : T\ \overline{v} \to \tau}$$

Figure 2: Typing rules for the core language of $F_C(X)$

$$\boxed{\Gamma \vdash decl : \Gamma'}$$

$$\text{Data}\;\dfrac{\overline{\Gamma \vdash_{\mathrm{TY}} \sigma : \star} \qquad \Gamma \vdash_{\mathrm{k}} \kappa : TY}{\Gamma \vdash \textbf{data } T : \kappa \textbf{ where } \overline{K : \sigma} : (T : \kappa, \overline{K : \sigma})}$$

$$\text{Type}\;\dfrac{\Gamma \vdash_{\mathrm{k}} \kappa : TY}{\Gamma \vdash (\textbf{type } S : \kappa) : (S : \kappa)} \qquad \text{Coerce}\;\dfrac{\Gamma \vdash_{\mathrm{k}} \kappa : CO}{\Gamma \vdash (\textbf{axiom } C : \kappa) : (C : \kappa)}$$

$$\boxed{\Gamma \vdash pgm : \sigma}$$

$$\text{Pgm}\;\dfrac{\begin{array}{cc}\overline{\Gamma \vdash cdecl : \Gamma_c} & \Gamma = \Gamma_0, \overline{\Gamma_c} \\ \overline{\Gamma \vdash decl : \Gamma_d} & \Gamma = \Gamma, \overline{\Gamma_d} \\ \multicolumn{2}{c}{\Gamma \vdash_{\mathrm{e}} e : \sigma}\end{array}}{\Gamma_0 \vdash \overline{cdecl}; \overline{decl}; e : \sigma}$$

Figure 3: Environment extension rules for $F_C(X)$

Theorem 4 (Domain-restricted reachability of types)**.** *Proof.* WRITE ME □

We then extend the typing judgements in Figure 2 with the two rules in Figure 4, and omit the other rules (transitivity etc.) since these can be easily figured out.

$$\text{ECoreCast}\;\dfrac{\Gamma \vdash_{\mathrm{e}} e : \zeta' \qquad \Gamma \vdash_{\mathrm{CO}} \gamma : \zeta' \underset{[T_1,T_2]}{\sim} \zeta}{\Gamma \vdash_{\mathrm{e}} e \underset{[T_1,T_2]}{\triangleright} \gamma : \zeta}$$

$$\text{CoCoreVar}\;\dfrac{\begin{array}{cc}\zeta_1 = (\mathtt{K}, \sigma_1) \in \Gamma & \zeta_2 = (\mathtt{K}, \sigma_2) \in \Gamma \\ \exists T_1, T_2. \left(\sigma_1 \underset{[T_1,T_2]}{\equiv} \sigma_2\right) & (\gamma : \zeta_1 \underset{[T_1,T_2]}{\sim} \zeta_2) \in \Gamma\end{array}}{\Gamma \vdash_{\mathrm{CO}} \gamma : \zeta_1 \underset{[T_1,T_2]}{\sim} \zeta_2}$$

$$\boxed{\Gamma \vdash cdecl : \Gamma'}$$

$$\text{CoreData}\;\dfrac{\overline{\Gamma \vdash_{\mathrm{TY}} \sigma : \star} \qquad \Gamma \vdash_{\mathrm{k}} \kappa : TY}{\Gamma \vdash \textbf{data extendable } T \;:\; \overline{\kappa} \textbf{ where } \overline{K : \sigma} : (T : \kappa, \overline{K : \sigma}, (T; \overline{(K, \sigma)}))}$$

$$\text{ExtData}\;\dfrac{\begin{array}{cc}\overline{\Gamma \vdash_{\mathrm{TY}} \sigma : \star} & \Gamma \vdash_{\mathrm{k}} \kappa : TY \\ \multicolumn{2}{c}{\overline{(K', \sigma') \triangleq \forall (K', \sigma'') \in lookup(T, \Gamma).(K', \sigma'')[\![T'/T]\!]}}\end{array}}{\Gamma \vdash \textbf{data } T' \;:\; \overline{\kappa} \textbf{ where } \overline{K : \sigma} \textbf{ extends}(\overline{T}) : (T' : \kappa, \overline{K : \sigma}, \overline{K' : \sigma'}, \overline{(K', \sigma') \underset{[T,T']}{\sim} (K', \sigma'')})}$$

$$\Xi \triangleq (T' : \kappa, \overline{K : \sigma}, \overline{K' : \sigma'}, \overline{(K', \sigma') \underset{[T,T']}{\sim} (K', \sigma'')})$$

$$\text{ExtData}\;\dfrac{\begin{array}{cc}\overline{\Gamma \vdash_{\mathrm{TY}} \sigma : \star} & \Gamma \vdash_{\mathrm{k}} \kappa : TY \\ \multicolumn{2}{c}{\overline{(K', \sigma') \triangleq \forall (K', \sigma'') \in lookup(T, \Gamma).(K', \sigma'')[\![T'/T]\!]}}\end{array}}{\Gamma \vdash \textbf{data extendable } T' \;:\; \overline{\kappa} \textbf{ where } \overline{K : \sigma} \textbf{ extends}(\overline{T}) : (\Xi, (T'; (K :: K', \sigma :: \sigma')))}$$

Figure 4: Extension of the typing rules to allow extension of data types

2.1.1 Insertion of Casts

In order to insert the casts correctly, we must do coverage analysis of the matching on the data types e.g.,

```
f :: T1 → T2
f ...
```

In this case we perform coverage analysis on `T1` and, when `f` fails to match on all of the data types, automatically inserts a default clause (if there is not already one) which behaves like the identity function, except that it inserts a cast $(K, \sigma) \underset{T_1,T_2}{\triangleright}$. We now formalise this argument.

Define the set of all constructors of an extended data type T coming from extended data types as C. Define the set of all specific constructors defined by T as D. Then we have that all of the constructors of T are $C \sqcup D$.

Now when performing coverage analysis of `f` we consider two cases – letting R be the covered cases and R' the non-covered cases:

- $R = C \sqcup D$: In this case, the programmer has manually coded a conversion between the two data types. Therefore, we do not have to insert any casts.

- $R \supsetneq D$. Then we add a default clause to `f` that performs a type coercion on the input.
- $R \subsetneq D$. In this case, it is an error, and we do not insert casts, since we cannot add a catchall clause to `f`.
- **It is worth noting that we could relax this requirement and add patterns for everything in $C \setminus R$ this would provide correctness right?**

3 Translation

In this section we go about detailing the source translation in order to implement these rules and to allow the use of extendable data types in Haskell.

3.0.2 Insertion of Casts

In order to insert the casts correctly, we must do coverage analysis of the matching on the data types e.g.,

```
f :: T1 → T2
f ...
```

In this case we perform coverage analysis on `T1` and, when `f` fails to match on all of the data types, automatically inserts a default clause (if there is not already one) which behaves like the identity function, except that it inserts a cast $(K, \sigma) \underset{T_1,T_2}{\rhd}$. We now formalise this argument.

Define the set of all constructors of an extended data type T coming from extended data types as C. Define the set of all specific constructors defined by T as D. Then we have that all of the constructors of T are $C \sqcup D$.

Now when performing coverage analysis of `f` we consider two cases – letting R be the covered cases and R' the non-covered cases:

- $R = C \sqcup D$: In this case, the programmer has manually coded a conversion between the two data types. Therefore, we do not have to insert any casts.
- $R \supsetneq D$. Then we add a default clause to `f` that performs a type coercion on the input.
- $R \subsetneq D$. In this case, it is an error, and we do not insert casts, since we cannot add a catchall clause to `f`.
- **It is worth noting that we could relax this requirement and add patterns for everything in $C \setminus R$ this would provide correctness right?**

4 Example

```
{-# LANGUAGE GADTs #-}
module Examples.Ex1 where


-- We start by defining the core ADT.
data extendable Core where
  Unit :: Core
  Var  :: Int  → Core
  Lam  :: Core → Core
  App  :: Core → Core → Core

-- We can then extend this core with other datatypes, note that even though
-- we treat the core as part of this data type (SExp) IT IS THE EXACT SAME
-- AS Core
data SExp where
    CallCC :: SExp → SExp
    Abort  :: SExp → SExp
  deriving (Show)
  extending (Core)

-- We can then extend this core with other datatypes, note that even though
-- we treat the core as part of this data type (TExp) IT IS THE EXACT SAME
```

```
-- AS Core
data TExp where
    Add :: TExp → TExp → TExp
  deriving (Show)
  extending (Core)

-- Since the core part is the exact same - "shared" - we no longer have
-- to convert every single part of the datatype when converting between the two:
convert :: SExp → TExp
convert (CallCC _ _) = Unit
convert (Abort _ _)  = Unit
convert t            = t

---------- What this should behave like ---------

data SExp where
    Unit :: SExp
    Var  :: Int  → SExp
    Lam  :: SExp → SExp
    App  :: SExp → SExp → SExp
    CallCC :: SExp → SExp
    Abort  :: SExp → SExp

data TExp where
    Unit :: TExp
    Var  :: Int  → TExp
    Lam  :: TExp → TExp
    App  :: TExp → TExp → TExp
    Add :: TExp → TExp → TExp

convert :: SExp → TExp
convert (CallCC _ _) = Unit
convert (Abort _ _)  = Unit
convert Unit = Unit
convert Var = Var
convert (Lam e) = Lam $ convert e
convert (App e1 e2)  = App (convert e1) (convert e2)
```

References

[1] Martin Sulzmann, Manuel MT Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System f with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66. ACM, 2007.