A Calculus of Definitions

1 Type theory

We describe how to implement a core type theory. This is very close to a functional programming language with λ abstraction and data types defined by constructors and functions defined by case on these data types. The difference with ordinary functional programming is that we can do *computation on types*.

The canonical *types* are either *dependent product* types or *labelled sums*.

The canonical elements are either λ-*abstraction* or *functions defined by case* or in *constructor form*.

We also have a *universe of small types*, with which we can do computation on types.

Like in any functional programming language we have a *let* (or *where*) construct, with which we can define elements by mutual recursion. It is possible in this language to define in a mutual recursive way (small) types and functions[1].

Interestingly, the language now looks very much like the language Lazy ML (one precursor of Haskell), where data types are also represented as labelled sums.

[1] We can represent *induction-recursion* in this way.

2 Some examples

```
N : U = 0 | S N

(/\) : N2 -> N2 -> N2 = (0 -> \ b -> 0 |1 -> (0 -> 0 |1 -> 1))

N2 : U = 0 | 1

N1 : U = 0

N0 : U = ()  -- empty labelled sum

T : N2 -> U = (0 -> N0 |1 -> N1)

neq : N -> N -> N2 =
 (0 -> (0 -> 1 |S m -> 0) |S n -> (0 -> 1 |S m -> neq n m))

mutual
 flist : U = Nil | Cons (a : N) (as : flist) (T (notin a as))

 notin : N -> flist -> N2 =
  \ a -> (Nil -> 1 |Cons b bs -> neq a b /\ notin a bs)

W (A : U) (B : A -> U) : U = Sup (a : A) (f : B a -> W A B)

tree (A : U) (B : A -> U) (C : (a : A) -> B a -> U)
     (d : (a : A) -> (b : B a) -> C a b -> A) (x : A) : U =
 Sup (y : B x) ((z : C x y) -> tree A B C d (d x y z))
```

3 Programming language

Programs

$$M, A \ ::= \ v_k \mid M\ M \mid \lambda M \mid \Pi\ A\ A \mid M\ D \mid c\ \vec{M} \mid B \mid L$$

Definitions, Branches and Labelled Sums

$$D \ ::= \ [\vec{M} : \vec{A}] \qquad B \ ::= \ c_1\ M_1, \ldots, c_k\ M_k \qquad L \ ::= \ c_1\ \vec{A}_1, \ldots, c_k\ \vec{A}_k$$

Environments, Contexts and Values

$$\rho \ ::= \ () \mid \rho, u \mid D\rho \qquad \Gamma \ ::= \ () \mid \Gamma, V$$

$$u, V \ ::= \ M\rho \mid u\ u \mid X_l \mid c\ \vec{u} \mid \Pi\ V\ V$$

Access rules

$$v_0(\sigma, u) = u \quad v_{k+1}(\sigma, u) = v_k\sigma$$

and if $\rho = [\vec{M} : \vec{A}]\sigma$ then

$$v_i\rho = v_i(\sigma, \vec{M}\rho)$$

Evaluation rules

$$(M_1\ M_2)\rho = M_1\rho\ (M_2\rho) \qquad (M\ D)\rho = M(D\rho)$$

$$(\Pi\ A\ F)\rho = \Pi\ (A\rho)\ (F\rho) \qquad (c\ \vec{M})\rho = c\ (\vec{M}\rho)$$

$$(\lambda\ M)\rho\ u = M(\rho, u) \qquad (c_1\ N_1, \ldots, c_k\ N_k)\rho\ (c_i\ \vec{u}) = N_i(\rho, \vec{u})$$

4 Type-checking rules

$$\frac{\rho, \Gamma \vdash_k A \qquad (\rho, X_k), (\Gamma, A\rho) \vdash_{k+1} A'}{\rho, \Gamma \vdash_k \Pi\ A\ (\lambda A')} \qquad \frac{\rho, \Gamma \vdash_k \vec{A_1} \quad \ldots \quad \rho, \Gamma \vdash_k \vec{A_n}}{\rho, \Gamma \vdash_k c_1\ \vec{A_1}, \ldots, c_n\ \vec{A_n}}$$

$$\frac{}{\rho, \Gamma \vdash_k () \to \rho, \Gamma, k} \qquad \frac{\rho, \Gamma \vdash_k A \qquad (\rho, X_k), (\Gamma, A\rho) \vdash_{k+1} \vec{A} \to \rho_1, \Gamma_1, l}{\rho, \Gamma \vdash_k A, \vec{A} \to \rho_1, \Gamma_1, l}$$

Rule for recursive definitions

$$\frac{\rho, \Gamma \vdash_k \vec{A} \to \rho_1, \Gamma_1, l \qquad \rho_1, \Gamma_1 \vdash_l \vec{M} : \vec{A}\rho}{\rho, \Gamma \vdash_k [\vec{M} : \vec{A}]}$$

Rules for elements

$$\frac{}{\rho, \Gamma \vdash_k () : ()\nu} \qquad \frac{\rho, \Gamma \vdash M : A\nu \qquad \rho, \Gamma \vdash \vec{M} : \vec{A}(\nu, M\rho)}{\rho, \Gamma \vdash_k M, \vec{M} : (A, \vec{A})\nu}$$

$$\frac{\rho, \Gamma \vdash_k N : \Pi\ V\ F \qquad \rho, \Gamma \vdash_k M : V}{\rho, \Gamma \vdash_k N\ M : F\ (M\rho)} \qquad \frac{(\rho, X_k), (\Gamma, V) \vdash_{k+1} N : F\ X_k}{\rho, \Gamma \vdash_k \lambda N : \Pi\ V\ F}$$

$$\frac{}{\rho, \Gamma \vdash_k v_n : \Gamma!n} \qquad \frac{\rho, \Gamma \vdash_k \vec{M} : \vec{A_i}\nu}{\rho, \Gamma \vdash_k c_i\ \vec{M} : L\nu}$$

$$\frac{(\rho, \vec{X_k}), \Gamma + \vec{X_k} : \vec{A_1}\nu \vdash_{k+l_1} N_1 : F\ (c\ \vec{X_k}) \ \ldots\ (\rho, \vec{X_k}), \Gamma + \vec{X_k} : \vec{A_n}\nu \vdash_{k+l_n} N_n : F\ (c\ \vec{X_k})}{\rho, \Gamma \vdash_k B : \Pi\ (L\nu)\ F}$$

$$\frac{\rho, \Gamma \vdash_k D \qquad D\rho, \Gamma + \vec{M}(D\rho) : \vec{A}\rho \vdash_{k+l} N : V}{\rho, \Gamma \vdash_k N\ D : V}$$

where $B = c_1\ N_1, \ldots, c_n\ N_n,\ L = c_1\ \vec{A_1}, \ldots, c_n\ \vec{A_n},\ D = [\vec{M} : \vec{A}],\ l = |\vec{A}|,\ l_i = |\vec{A_i}|$

$$\Gamma + () : ()\nu = \Gamma \qquad \Gamma + u, \vec{u} : (A, \vec{A})\nu = \Gamma, A\nu + \vec{u} : \vec{A}(\nu, u)$$

We can add a universe U of small types with computation rules $U\rho = U$.

5 Reification

Each branch B has a name f_B and each labelled sum L a name d_L associated to it.

$$R_k\ X_l = v_{k-l-1} \qquad R_k\ ((\lambda M)\rho) = \lambda R_{k+1}(M(\rho, X_k)) \qquad R_k\ (u_1\ u_2) = R_k\ u_1\ (R_k\ u_2)$$

$$R_k\ (\Pi\ V\ F) = \Pi\ (R_k\ V)\ (R_k\ F) \qquad R_k\ (c\ \vec{u}) = c\ (R_k\ \vec{u})$$

$$R_k\ (B\rho) = f_B(R_k\ \rho) \qquad R_k\ (L\rho) = d_L(R_k\ \rho)$$

$$R_k\ () = () \qquad R_k\ (\rho, u) = (R_k\ \rho, R_k\ u) \qquad R_k\ (D\rho) = R_k\ \rho$$

6 Projection and conversion

In order to get η-conversion, we introduce the projection functions

$$
\begin{array}{lcl}
\mathsf{p}\ L\rho\ (c\ \vec{u}) & = & c\ (\mathsf{q}\ \vec{A}\rho\ \vec{u}) \quad \text{with} \quad c\ \vec{A}\ \text{ in } \ L \\
\mathsf{p}\ L\rho\ k & = & k \\
\mathsf{p}\ (\Pi\ a\ f)\ w & = & x \mapsto \mathsf{p}\ (f(\mathsf{p}\ a\ x))\ (w(\mathsf{p}\ a\ x)) \\
\mathsf{p}\ \mathsf{U}_j\ L\rho & = & L\rho \\
\mathsf{p}\ \mathsf{U}_j\ (\Pi\ a\ f) & = & \Pi\ (\mathsf{p}\ \mathsf{U}_j\ a)\ ((\mathsf{p}\ \mathsf{U}_j) \circ\ f\ \circ (\mathsf{p}\ a)) \\
\mathsf{p}\ \mathsf{U}_j\ \mathsf{U}_i & = & \mathsf{U}_i \qquad\qquad \text{if} \quad i < j \\
\mathsf{p}\ \mathsf{U}_j\ k & = & k \\
\mathsf{p}\ k\ k' & = & k'
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{q}\ ()\rho\ () & = & () \\
\mathsf{q}\ (A, \vec{A})\rho\ (u, \vec{u}) & = & v, \mathsf{q}\ \vec{A}(\rho, v)\ \vec{u} \ \text{ where } \ v = \mathsf{p}\ (A\rho)\ u
\end{array}
$$

and when introducing a fresh value v_k of type $A\rho$ we use $\mathsf{p}\ (A\rho)\ v_k$ instead.

7 Infinite structures

The main idea is to use *closure* to represent infinite structures. We use it already to represent recursive data types and recursively defined functions, and we use it now to represent *streams*.

A first attempt would be to have a notion of "lazy" constructors so that $(c\ M)\rho$ is canonical. For a "strict" constructor $(c\ M)\rho$ reduces to $c\ (M\rho)$. From the user point of view, each constructor is used in a "generative" way. This means that if we define

$$\omega = s\ \omega \qquad\qquad \omega_1 = s\ \omega_1$$

then ω and ω_1 are not convertible.

However it is then not possible to give good sense of dependent case. For instance if we define the type $\Omega = s\ \Omega$ and $\omega : \Omega$ by $\omega = s\ \omega$ then we cannot typecheck

$$f : (x : \Omega) \to C(x) \qquad\quad f\ (s\ y) = b$$

since b should have type $C(s\ y)$.

8 New terms

We extend the syntax of our language with

$$M, A \ ::= \ l_1 : A_1, \dots, l_n : A_n \mid l_1 = M_1, \dots, l_n = M_n \mid M.l$$

with the new computation rules

$$(M.l)\rho = M\rho.l \qquad\qquad (l_1 = M_1, \dots, l_n = M_n)\rho.l_i = M_i\rho$$

and the new typing rules

$$\frac{\rho, \Gamma \vdash_k A_1 \ \dots \ \rho, \Gamma \vdash_k A_n}{\rho, \Gamma \vdash_k (l_1 : A_1, \dots, l_n : A_n)}$$

$$\frac{\rho, \Gamma \vdash_k M_1 : A_1\nu \ \dots \ \rho, \Gamma \vdash_k M_n : A_n\nu}{\rho, \Gamma \vdash_k (l_1 = M_1, \dots, l_n = M_n) : (l_1 : A_1, \dots, l_n : A_n)\nu} \qquad \frac{\rho, \Gamma \vdash_k M : (l_1 : A_1, \dots, l_n : A_n)\nu}{\rho, \Gamma \vdash_k M.l_i : A_i\nu}$$

This is a good modularity test. We don't have to change the other clauses (in particular the clauses for checking recursive definitions).

9 Examples

We can define the type of streams $stream\ A = (hd : A, tl : stream\ A)$. The constant stream 0 of type $stream\ N$ would be $0s = (hd = 0, tl = 0s)$. Then $0s.tl$ and $0s$ are convertible.

One can define $cons\ a\ as = (hd = a, tl = as)$. But notice then that if we define $us = cons\ 0\ us$ then the semantics of us is $\perp$. So the function $cons$ cannot be used to define the constant stream 0.

This corresponds to the syntactical fact that the normal form of $0s$ is finite and is

$$(hd = 0, tl = 0s)D$$

where D is the definition $0s = (hd = 0, tl = 0s)$ while the normal for of us would be the infinite expression

$$(hd = a, tl = as)(a = 0, as = (hd = a, tl = as)(a = 0, as = (hd = a, tl = as)(a = 0, as = \ldots)))$$