A Calculus of Definitions

March 18, 2008

The mathematicians will probably raise objections against that, because contemporary mathematics is thoroughly extensional and hence no clear notions of intensions have been developed. But it is nevertheless certain that, at least within the framework of a particular language, completely precise concepts of this kind can be defined. (Gödel, letter to Bernays, 1970)

Introduction

1 Programs

The programs are usual (untyped) λ-expressions with pairs. We add constants and definitions by cases, like in the language PCF

$$M \ ::= \ x \mid \lambda x.M \mid M \ M \mid M, M \mid M.1 \mid M.2 \mid c \mid M \ D \mid B$$

the category B is for *functions defined by cases*

$$B \ ::= \ c_1 \to M_1, \ \ldots, \ c_k \to M_k$$

and the category D is for *mutual recursive definitions*

$$D \ ::= \ x_1 = M_1, \ \ldots, \ x_k = M_k$$

Notice that programs are first-order objects, with an associated decidable equality.

2 Denotational semantics

We take the Scott domain solution of the recursive equation

$$\mathsf{V} \ = \ [\mathsf{V} \to_s \mathsf{V}] + \mathsf{V} \times_s \mathsf{V} + \mathsf{Ide}$$

where the sum is the *non* lifted sum. Any element of V different from $\perp$ is thus a *strict* function f or of the form c or is a pair u_1, u_2 of two elements $\neq\perp$.

One essential point is to consider a *strict* semantics for our language.

We define $[\![M]\!]_\rho$ in V for ρ an environment, i.e. a function from variables to V.

We introduce first a function *app* in $\mathsf{V} \to \mathsf{V} \to \mathsf{V}$ by taking *app* d_1 d_2 to be $\perp$ if d_1 is not a function; if d_1 is a function f and *app* d_1 d_2 is defined to be $f(d_2)$. We define then as usual $[\![M_1 \ M_2]\!]_\rho$ to be *app* $[\![M_1]\!]_\rho$ $[\![M_2]\!]_\rho$. We define also $[\![x]\!]_\rho = \rho(x)$.

We define $[\![\lambda x.M]\!]_\rho$ to be the function f such that $f(\perp) = \perp$ and if $u \neq \perp$

$$f(u) = [\![M]\!]_{\rho, x=u}$$

Similarly, $[\![B]\!]_\rho$ is the function f such $f(c) = [\![N]\!]_\rho$ if $c \to N$ is in B and $f(c,u) = app\ [\![N]\!]_\rho\ u$ if $c \to N$ is in B[1]. In all the other cases we have $f(v) = \bot$.

We take $[\![M_1, M_2]\!]_\rho = ([\![M_1]\!]_\rho, [\![M_2]\!]_\rho)$ if both $[\![M_i]\!]_\rho \neq \bot$ and $[\![M_1, M_2]\!]_\rho = \bot$ if $[\![M_1]\!]_\rho = \bot$ or $[\![M_2]\!]_\rho = \bot$. We take $[\![c]\!]_\rho = c$.

Finally let D be a definition $x_1 = M_1,\ \ldots,\ x_k = M_k$. We define

$$[\![M\ D]\!]_\rho = [\![M]\!]_{\rho'}$$

where the environment ρ' is the least solution of the recursive equation

$$\rho' = (\rho, x_1 = [\![M_1]\!]_{\rho'},\ \ldots,\ x_k = [\![M_k]\!]_{\rho'})$$

3 Definitional Equality

The goal of this section is to introduce the notion of definitional equality or convertibility between programs. We want to represent the idea that two terms are convertible if we can get to the same term by unfolding some definitions.

The equality $[\![M_1]\!] = [\![M_2]\!]$ is not decidable, and does not really coincide with the notion of "having the same definition". To take a simple example, if we define

$$M_1 = zero_1(zero_1 = (0 \to 0, S \to zero_1))$$

$$M_2 = zero_2(zero_2 = (0 \to 0, S \to (0 \to 0, S \to zero_2)))$$

we do have $[\![M_1]\!] = [\![M_2]\!]$ but we cannot say that M_1 and M_2 are convertible, in the sense of having the same definition[2].

We are going to define a notion of definitional equality, which is *decidable* over programs M such that $[\![M]\!] \neq \bot$ (intuitively, programs that "make sense"). For this, we introduce the notion of *values*, which is convenient to represent what happens when we unfold a definition. Furthermore, this notion is closely connected to what happens in computations of programs with environment machines. Like in any environment machines, the programs are essentially static objects and the computations are done on values. The values can be described in the following way

$$u\ ::=\ X_i \mid u, u \mid u\ u \mid c \mid u.1 \mid u.2 \mid M\sigma, \qquad \sigma\ ::=\ () \mid \sigma, x = u \mid D\sigma$$

The values X_i, called *generic* values, are introduced here only to define the notion of normalisable values (see below). The equality on values in not decidable, and is given by the following (weak) conversion rules

$$(M_1\ M_2)\sigma = M_1\sigma\ (M_2\sigma), \qquad (\lambda x.M)\sigma\ u = M(\sigma, x = u),$$

$$c\sigma = c, \qquad (M_1, M_2)\sigma = (M_1\sigma, M_2\sigma), \qquad (u_1, u_2).i = u_i$$

$$(M\ D)\sigma = M(D\sigma), \qquad (B\sigma)\ c = M\sigma, \qquad (B\sigma)\ (c, u) = M\sigma\ u$$

[1] This implicit anticurrification is quite useful in the representation of type theory in our language.

[2] Intuitively, both terms represent the infinite object

$$(0 \to 0, S \to (0 \to 0, S \to (0 \to 0, \ldots)))$$

It might be that bissimulation on these kind of objects is actually decidable, but this would not capture the notion of definitional equality.

where in the last two rules it is assumed that $c \to M$ occurs in B. The remaining rules are access rules

$$x(\sigma, x = u) = u, \qquad x(\sigma, y = v) = x\sigma \quad (x \neq y)$$

and $x(D\sigma) = x\sigma$ if x is not declared in D and (the main rule)

$$x(D\sigma) = M(D\sigma)$$

if $x = M$ occurs in D.

It is rather direct that, if we see these conversion rules as a rewriting system, we get a confluent system.

For instance, if $B = (0 \to x \mid S \to \lambda y.(S, add\ x\ y))$ and D is $add = \lambda x.B$ we have the following computation

$$(add\ 0\ (S,0))\ D = B(D, x = 0)\ (S,0) = (S, B(D, x = 0)\ 0) = (S,0)$$

while, if D_1 is the definition $x = (S, x)$ we have the computation

$$xD_1 = (S,x)\ D_1 = (S, xD_1) = (S, (S, xD_1)) = \ldots$$

The *strong* conversion is obtained by adding the ξ-rule that $(\lambda x_1.M_1)\sigma_1$ and $(\lambda x_2.M_2)\sigma_2$ are convertible whenever $M_1(\sigma_1, x_1 = X_i)$ and $M_2(\sigma_2, x_2 = X_i)$ are, where X_i is a "fresh" generic value, i.e. a value which does not occur neither in σ_1 nor in σ_2. (It is arguable if this step corresponds really to definitional equality.)

4 Head Normalisable values

We define the head reduction relation $u \succ u_1$ on values. The rules are the following

$$(M_1\ M_2)\sigma \succ M_1\sigma\ (M_2\sigma) \qquad (\lambda x.M)\sigma\ u \succ M(\sigma, x = u) \qquad c\sigma \succ c$$

$$(M_1, M_2)\sigma \succ (M_1\sigma, M_2\sigma) \qquad (u_1, u_2).i \succ u_i \qquad (M.i)\sigma \succ (M\sigma).i$$

$$(M\ D)\sigma \succ M(D\sigma) \qquad (B\sigma)\ c \succ N\sigma \quad (B\sigma)\ (c,u) \succ N\sigma\ u \quad (c\ \vec{x} \to N \quad in \quad B)$$

$$x(\sigma, x = u) \succ u \qquad x(\sigma, y = v) \succ x\sigma \qquad (x \neq y)$$

$$x(D\sigma) \succ x\sigma \quad (x\ not\ in\ D) \qquad x(D\sigma) \succ M(D\sigma) \quad (x = M\ in\ D)$$

$$\frac{v \succ v_1}{v\ u \succ v_1\ u} \qquad \frac{u \succ u_1}{B\sigma\ u \succ B\sigma\ u_1}$$

We define next when a value u is HN. First it has to have a *canonical form* $u \succ^* v$ which has to be of the following form

$$v\ ::=\ (\lambda x.M)\sigma \mid c \mid u, u \mid B\sigma \mid k \qquad k\ ::=\ X_i \mid k\ u \mid B\sigma\ k \mid k.1 \mid k.2$$

and which has to satisfy

if $v = (\lambda x.M)\sigma$ then $M(\sigma, x = X_i)$ should be HN
if $v = B\sigma$ then σ should be HN
if $v = k\ v'$ then v' and k should be HN
if $v = B\sigma\ k$ then k and σ should be HN
if $v = v_1, v_2$ then v_1 and v_2 should be HN
if $v = c$ then it is HN

if $v = X_i$ then it is HN

and we define σ HN by

if $\sigma = (\sigma_1, x = v)$ then v and σ_1 should be HN
if $\sigma = D\sigma_1$ then σ_1 should be HN
if $\sigma = ()$ then it is HN

For instance x $(x = (S, x))$ has no canonical form while $add\ D$, where D is $add = \lambda x.B$ and B is $(0 \to x \mid S \to \lambda y.(S, add\ x\ y))$ is HN, since it has a canonical form $(\lambda x.B)D$ and $B(D, x = X_i)$ is HN.

Proposition 4.1 *Strong convertibility is decidable on HN values.*

5 Main result

Theorem 5.1 *If M is a closed program and $[\![M]\!] \neq \bot$ then the value $M()$ is HN.*

6 Representation of Gödel system T

We can embed Gödel system T in the present calculus by using the definition D

$$rec\ =\ \lambda f.\lambda a.(0 \to a,\ S \to \lambda x.f\ x\ (rec\ f\ a\ x))$$

Usually, one defines (the untyped version of) Gödel system T by extending untyped λ-calculus with constants 0, S, rec and conversion equations

$$rec\ f\ a\ 0 = a, \quad rec\ f\ a\ (S\ x) = f\ x\ (rec\ f\ a\ x)$$

Proposition 6.1 *Let M_1, M_2 be two terms of Gödel system T. They are convertible, as terms of Gödel system T iff the values M_1D and M_2D are convertible.*

This shows that this representation is faithful. We can represent in our calculus any system containing functions defined by case analysis. In our calculus we can define *locally* such systems, or we can have such systems depending on parameters, which is essential for modularisation.

Furthermore Theorem 5.1 provides a sufficient condition ensuring normalisation (and hence decidability of convertibility). As usual, we can prove $[\![M]\!] \neq \bot$ by introducing *totality subset* on the domain V, that are non empty subset of V not containing $\bot$.

7 Proof of Main Theorem

For the proof of Theorem 5.1 we formulate the semantics $[\![M]\!]_\rho$ in V as a typing relation $\Gamma \vdash M : U$ where the "types" are finite elements of V. The finite elements $\neq \bot$ of V are of the form

$$W\ ::=\ \nabla \mid c \mid W, W \mid W \to W \mid W \cap W$$

We introduce the formal relation of inclusion $U_1 \subseteq U_2$ between these elements which corresponds to the opposite of the relation in the domain V. The elements U are also called *formal neighbourhood*. We have also an operational interpretation $[U]$ of each formal neighbourhood. This is a set of HN values defined in such a way that $[U_1] \subseteq [U_2]$ if $U_1 \subseteq U_2$, We have also that $[\nabla]$ is the set of all neutral values. What is important is that the formal inclusion relation is *decidable*. The element ∇ correspond to the top element $\top$ of the domain V.

We define $[W]$ by induction on W.

$[\nabla]$ is the set of neutral values

$[c]$ is the set of HN values, neutral or of canonical form c

$[U \to V]$ is the set of HN values w, neutral or of canonical form $(\lambda x.N)\sigma$ or $B\sigma$, and such that $w\ u$ is in $[V]$ if u is in $[U]$

$[U_1, U_2]$ is of HN values, neutral or of canonical form u_1, u_2 and such that u_i is in $[U_i]$

$[U_1 \cap U_2]$ is $[U_1] \cap [U_2]$

We may write $u \in U$ instead of $u \in [U]$.

The typing rules are the following, where $\Gamma(x)$ represents A such that $x : A$ appears last in Γ (this notation requires that x is declared in Γ).

$$\frac{\Gamma \vdash M : U \quad U \subseteq V}{\Gamma \vdash M : V} \qquad \frac{\Gamma \vdash M : U_1 \qquad \Gamma \vdash M : U_2}{\Gamma \vdash M : U_1 \cap U_2}$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma, x : U \vdash N : V}{\Gamma \vdash \lambda x.N : U \to V} \qquad \frac{\Gamma \vdash N : U \to V \quad \Gamma \vdash M : U}{\Gamma \vdash N\ M : V}$$

$$\frac{\Gamma \vdash M_1 : U_1 \qquad \Gamma \vdash M_2 : U_2}{\Gamma \vdash M_1, M_2 : (U_1, U_2)} \quad \frac{\Gamma \vdash M : (U_1, U_2)}{\Gamma \vdash M.i : U_i} \quad \frac{}{\Gamma \vdash c : c}$$

If $c \to N$ appears in B we have

$$\frac{\Gamma \vdash N : V}{\Gamma \vdash B : c \to V} \qquad \frac{\Gamma \vdash N : U \to V}{\Gamma \vdash B : (c, U) \to V}$$

Finally we have the typing rule for $\Gamma \vdash MD : U$. This has the following form.

$$\frac{\Gamma^{(0)} \vdash M_1 : U_1^{(1)} \ \ldots\ \Gamma^{(0)} \vdash M_k : U_k^{(1)} \ \ldots\ \Gamma^{(l-1)} \vdash M_1 : U_1^{(l)} \ \ldots\ \Gamma^{(l-1)} \vdash M_k : U_k^{(l)} \quad \Gamma^{(l)} \vdash M : U}{\Gamma \vdash MD : U}$$

where D is $x_1 = M_1, \ldots, x_k = M_k$ and $\Gamma^{(j)}$ is $\Gamma, x_1 : U_1^{(j)}, \ldots, x_k : U_k^{(j)}$ and $U_i^{(0)}$ is Δ. This rule reflects the fact that the semantics $[\![MD]\!]_\rho$ is $[\![M]\!]_{\rho'}$ where the environment ρ' is the least solution of the recursive equation $\rho' = (\rho, x_1 = [\![M_1]\!]_{\rho'},\ \ldots,\ x_k = [\![M_k]\!]_{\rho'})$.

We generalise the statement of Theorem 5.1 in the following way.

Lemma 7.1 *If $\Gamma \vdash M : U$, σ is HN and $x\sigma$ is in $[\Gamma(x)]$ for each variable x declared in Γ then $M\sigma$ is in $[U]$.*

Proof. The proof is by induction on the proof of $\Gamma \vdash M : U$.

The result is clear for the first two rules, and for the variable rule.

For the abstraction rule: if σ is HN and $x\sigma$ is in $[\Gamma(x)]$ for each variable x declared in Γ and U is $\neq \Delta$ and $u \in U$ and we assume $N(\sigma, x = u)$ in V then $(\lambda x.N)\sigma\ u$ is in V since it head reduces to $N(\sigma, x = u)$

For the application rule: if $N\sigma$ is in $U \to V$ and $M\sigma$ is in U then $N\sigma\ (M\sigma)$ and so $(N\ M)\sigma$ is in V.

The constant rule is clear.

For the case rule: we assume that $c \to N$ appears in B and $\Gamma \vdash N : V$. We want to show that $B\sigma$ is in $c \to V$. Since σ is HN we only have to show that $B\sigma\ c$ is in V. This is the case since $B\sigma\ c \succ N\sigma$ and $N\sigma$ is in V by induction. If we have $\Gamma \vdash N : U \to V$ we show that $B\sigma$ is in $(c, U) \to V$. Since σ is HN we only have to show that $B\sigma\ (c, u)$ is in V. This is the case since $B\sigma\ (c, u) \succ N\sigma\ u$ and $N\sigma$ is in $U \to V$ by induction.

For the where rule. If we assume that we have $x\sigma$ in $[\Gamma(x)]$ for all x declared in Γ then we prove by induction that $xD\sigma$ is in $[\Gamma^{(j)}(x)]$ for all $j \leqslant l$ and so $M(D\sigma)$ is in $[U]$ and hence also $(MD)\sigma$ is in $[U]$. (Notice that $D\sigma$ is HN if σ is HN.) □

8 Encoding of type theory

8.1 Denotational semantics

We describe first the encoding at the level of denotational semantics.

We suppose that the discrete domain Ide contains two special elements Π and Σ and that the other elements are either simple $c, i, j, \ldots$ or encode a finite set of simple identifiers $I, J, \ldots$

We write $\Pi\ a\ f$ instead of $(\Pi, (a, f))$ and similarly $\Sigma\ a\ f$ instead of $(\Sigma, (a, f))$.

A *totality* on V is a subset $X \subseteq \mathsf{V}$ such that $\bot \notin X$ and $\top \in X$. We write $\mathsf{TP}(\mathsf{V})$ the set of all totality on V.

An *interpretation* of type theory is a pair (X, El) with X in $\mathsf{TP}(\mathsf{V})$ and El in $X \to \mathsf{TP}(\mathsf{V})$ such that $El(\top)$ is the singleton $\{\top\}$ (we have $\top$ in X since X is a totality) and we have I in X and $El(I) = \{\top\} \cup I$.

The set of all interpretations $\mathcal{I}$ is ordered by the relation: $(X, El) \leqslant (X', El')$ iff $X \subseteq X'$ and El' extends El. This forms a conditionally complete poset and it has a least element X_0, El_0 where X_0 is the set of all elements I and $\top$.

We define a *monotone* function $\Psi : \mathcal{I} \to \mathcal{I}$. If $(X_1, El_1) = \Psi(X, El)$ the intuition is that the elements of X_1 are *productive* trees where the nodes are products or sums, the branching given by elements $El(a)$ with a in X, and the leaves are of the form I. More formally, X_1 is the *greatest* subset of V such that b is in X_1 iff

$b = I$ or

$b = \Pi\ a\ f$ and $a \in X$ and $f\ u$ in X_1 for all u in $El(a)$ or

$b = \Sigma\ a\ f$ and $a \in X$ and $f\ u$ in X_1 for all u in $El(a)$.

We define then $v \in El_1(b)$ *inductively* by the clauses

$\top \in El_1(b)$ and

$i \in El_1(I)$ if i is in I and

$v \in El_1(\Pi\ a\ f)$ if $v\ u \in El_1(f\ u)$ for all u in $El(a)$ and

$(u, v) \in El_1(\Sigma\ a\ f)$ if u in $El(a)$ and $v \in El_1(f\ u)$.

We have the empty type $N_0 = \{\}$ and the unit type $N_1 = \{\cdot\}$ and the Boolean type $N_2 = \{0, 1\}$. These are in X_0.

For instance we encode *nat* as the least fixed-point of the equation $nat = \Sigma\ I\ f$ with $I = \{0, S\}$ and $f\ 0 = N_1$ and $f\ S = nat$. We have *nat* in X_1 where $X_1, El_1 = \Psi(X_0, El_0)$ and $El_1(nat)$ contains $(0, \cdot),\ (S, (0, \cdot)), (S, (S, (0, \cdot))), \ldots$

Theorem 8.1 *The least fixed-point* (Typ, El) *of the operator* Ψ *(which exists since* $\mathcal{I}$ *is conditionally complete and has a least element) is closed under product and sum.*

This interpretation Typ contains most of the usual data type we need for type theory. It contains all finite types (obtained at level 0), all first-order types (obtained at level 1), the type of ordinal numbers (obtained at level 2), ... We can define $a + b = \Sigma\ N_2\ f$ with $f\ 0 = a$ and $f\ 1 = b$ and we have that Typ is closed under disjoint sum. It is also closed by list formation: if a is in Typ then the type $[a]$ of lists over a is also in Typ. We even that Typ contains the least-fixed point of the equation $a = \Sigma\ N_1\ f$ with $f\ 0 = [a]$.

A question is whether Typ contains the universe defined by the recursive equations

$$U = \Sigma\ N_2\ f, \qquad T = (0 \to nat, 1 \to \lambda(a, f).(\Pi x : T\ a)T\ (f\ x))$$

with $f\ 0 = N_1$ and $f\ 1 = (\Sigma a : V)T\ a \to V$.

More simply, one can ask if Typ contains the type recursively defined by

$$S = \Sigma N_2\ f$$

with $f\ 0 = N_1$ and $f\ 1 = (\Sigma x : S)S$.

8.2 Syntactical representation

We introduce the syntactic sugar $(c_1\ A_1 \mid \ldots \mid c_k\ A_k)$ for $\Sigma\ I\ (c_1 \to A_1, \ldots, c_k \to A_k)$. We write N_1 for the enumeration type $\{\cdot\}$ and N_0 for the enumeration type $\{\}$. We write also simply for instance $(c_1\ A_1 \mid c_2)$ if A_2 is N_1. We can then define $nat = (0 \mid S\ nat)$. We write $c\ M$ instead of (c, M) and c instead of $(c, \cdot)$. We have then $0 : nat$ and $S\ M : nat$ if $M : nat$.

We represent in this way type theory. Notice that if $[\![A]\!] \in \mathsf{Typ}$ and $[\![a]\!] \in El([\![A]\!])$ then we have $[\![A]\!] \neq \bot$ and $[\![a]\!] \neq \bot$ and hence A and a are HN.

So we have *decidable conversion* as long as we form only expressions semantically justified by Typ, El.

9 Representation of infinite objects

We have a type of streams

$$S\ A = \Pi\ I\ (hd \to A,\ tl \to S\ A)$$

where $I = \{hd, tl\}$. The semantics of $S\ A$ is in Typ if the semantics of A is. However with the definition of $El\ (S\ A)$ as well-founded trees, then this semantics will be empty.

It makes sense however to consider the productive elements of this type, which contains for instance the semantics of

$$as = (hd \to a, tl \to as)$$

if a is in $El(A)$.

10 Implementation in Haskell

We use a nameless representation and using sigma types, we can replace mutual recursive definition by one recursive definition. We can also merge the type of expression and the type of environment. We obtain in this way the following implementation.

```
-- nameless miniTT, with recursive definitions

module Enum1 where

type Brc = [(String,Exp)]

type Name = String

data Exp =
   Comp Exp Exp
 | App Exp Exp
 | Pi Exp Exp
```

```
 | Sig Exp Exp
 | Pair Exp Exp
 | Lam Exp
 | Fst Exp
 | Snd Exp
 | Var Int              -- de Bruijn level or generic values
 | Ref Int              -- de Bruijn index
 | Def Exp Exp          -- unit substitutions
 | Fun Brc
 | Con Name
 | Enum [Name]
       deriving (Show,Eq)

eval :: Exp -> Exp -> Exp       -- eval is also composition!

eval (Comp t1 t2) s = eval t1 (eval t2 s)
eval (Pair t1 t2) s = Pair (eval t1 s) (eval t2 s)
eval (App t1 t2) s = app (eval t1 s) (eval t2 s)
eval (Pi a b) s = Pi (eval a s) (eval b s)
eval (Sig a b) s = Sig (eval a s) (eval b s)
eval (Fst t) s = fstE (eval t s)
eval (Snd t) s = sndE (eval t s)
eval (Ref k) s = getE k s
eval e@(Con _) s = e
eval e@(Enum _) s = e
eval t s = Comp t s

app :: Exp -> Exp -> Exp
app (Comp (Lam b) s) u = eval b (Pair s u)
app (Comp (Fun ces) s) (Con c) = eval (get c ces) s
app (Comp (Fun ces) s) (Pair (Con c) u) = app (eval (get c ces) s) u
app f u = App f u

getE 0 s@(Comp (Def m _) a) = eval m s
getE 0 (Pair _ u) = u
getE (k+1) (Comp _ s) = getE k s
getE (k+1) (Pair s _) = getE k s

fstE (Pair u1 u2) = u1
fstE u = Fst u

sndE (Pair u1 u2) = u2
sndE u = Snd u

data G a = Success a | Fail Name

instance  Monad G  where
    (Success x) >>= k     =  k x
    Fail s   >>= k   =  Fail s
```

```
    return           =  Success
    fail             =  Fail

eqG s1 s2 | s1 == s2 = return ()
eqG s1 s2            = Fail ("eqG " ++ show s1 ++ " =/= " ++ show s2)

check :: Int -> Exp -> [Exp] -> Exp -> Exp -> G ()
checkT :: Int -> Exp -> [Exp] -> Exp -> G ()
checkI :: Int -> Exp -> [Exp] -> Exp -> G Exp
checkD :: Int -> Exp -> [Exp] -> Exp -> G (Exp,[Exp])

checkD k rho gam d@(Def m a) =
 do
  checkT k rho gam a
  check (k+1) (Pair rho (Var k)) (v:gam) v m
  return (Comp d rho,v:gam)
   where v = eval a rho
checkD k rho gam u = Fail ("checkD " ++ show u)

checkT k rho gam t = case t of
 Con "U" -> return ()
 Enum _ -> return ()
 Pi a (Lam b) -> do
                   checkT k rho gam a
                   checkT (k+1) (Pair rho (Var k)) ((eval a rho):gam) b
 Sig a (Lam b) -> do
                   checkT k rho gam a
                   checkT (k+1) (Pair rho (Var k)) ((eval a rho):gam) b
 Pi (Enum _) (Fun es) -> sequence_ [checkT k rho gam e | (_,e) <- es]
 Sig (Enum _) (Fun es) -> sequence_ [checkT k rho gam e | (_,e) <- es]
 _ -> checkI k rho gam t >>= eqG (Con"U")

upd k rho = Pair rho (Var k)

check k rho gam a t = case (a,t) of
 (_,Con c) -> extEnG c a
 (Con"U",Pi a (Lam b)) -> do
                      check k rho gam (Con"U") a
                      check (k+1) (upd k rho) ((eval a rho):gam) (Con"U") b
 (Con"U",Sig a (Lam b)) -> do
                       check k rho gam (Con"U") a
                       check (k+1) (upd k rho) ((eval a rho):gam) (Con"U") b
 (Con"U",Pi (Enum _) (Fun es)) -> sequence_ [check k rho gam (Con"U") e | (_,e) <- es]
 (Con"U",Sig (Enum _) (Fun es)) -> sequence_ [check k rho gam (Con"U") e | (_,e) <- es]
 (Pi (Enum en) f,Fun es) -> sequence_ [check k rho gam (app f (Con i)) e | (i,e) <- es]
 (Pi a f,Lam t) -> check (k+1) (upd k rho) (a:gam) (app f (Var k)) t
 (Sig a f,Pair m1 m2) -> do
                         check k rho gam a m1
                         check k rho gam (app f (eval m1 rho)) m2
```

```
 (Pi (Sig (Enum en) f) g,Fun es) ->
   if map fst es == en
     then sequence_ [check (k+1) (upd k rho) ((app f (Con i)):gam)
                           (app g (Pair (Con i) (Var k))) e
                     | (i,Lam e) <- es]
     else fail ("case branches does not match the data type ")
 (a,Fun es) -> Fail ("checkFun " ++ show a)
 (_,Comp t d) -> do
                  (rho1,gam1) <- checkD k rho gam d
                  check k rho1 gam1 a t
 _ -> do
       a' <- checkI k rho gam t
       eqG (reif k a) (reif k a')

checkI k rho gam e = case e of
 Ref k -> return (gam !! k)
 Fst t -> do
           c <- checkI k rho gam t
           (a,b) <- extSig c
           return a
 Snd t -> do
           c <- checkI k rho gam t
           (a,f) <- extSig c
           return (app f (fstE (eval t rho)))
 App n m -> do
             c <- checkI k rho gam n
             (a,f) <- extPi c
             check k rho gam a m
             return (app f (eval m rho))
 Comp t d -> do
              (rho1,gam1) <- checkD k rho gam d
              checkI k rho1 gam1 t
 Enum _ -> return (Con"U")
 _ -> Fail ("checkI " ++ show e)

extPi (Pi a b) = return (a,b)
extPi t = Fail (show t ++ " is not a product")

extSig (Sig a b) = return (a,b)
extSig t = Fail (show t ++ " is not a sigma")

extEnG c (Enum cs) =
 if elem c cs then return () else Fail ("extEnG " ++ c ++ " " ++ show cs)
extEnG c a = Fail (show a ++ " is not an enumeration type")

reif :: Int -> Exp -> Exp        -- reify function
reif k (App e1 e2) = App (reif k e1) (reif k e2)
reif k (Pair e1 e2) = Pair (reif k e1) (reif k e2)
reif k (Pi e1 e2) = Pi (reif k e1) (reif k e2)
```

```
reif k (Sig e1 e2) = Sig (reif k e1) (reif k e2)
reif k (Fst e) = Fst (reif k e)
reif k (Snd e) = Snd (reif k e)
reif k f@(Comp (Lam _) _) = Lam (reif (k+1) (app f (Var k)))
reif k (Comp e r) = Comp e (reif k r)
reif k (Var l) = Ref (k-l-1)
reif k e@(Con _)  = e
reif k e@(Enum _)  = e

get s [] = error ("get " ++ show s)     -- should never occur
get s ((s1,u):us) | s == s1 = u
get s ((s1,u):us)           = get s us

getG s [] = Fail ("getG " ++ show s)     -- should never occur
getG s ((s1,u):us) | s == s1 = return u
getG s ((s1,u):us)           = getG s us
```