1 Grammars

In the following, D is a data constructor, f a function symbol and we consider them as strings. n represents an integer.

1.1 λ-lifted haskell subset

$$\begin{array}{rcll}
u, e & := & x \mid f \mid e\ e \mid D(e, \ldots, e) \mid \texttt{BAD} \mid n & (\textit{Expression}) \\
p & := & \Delta, p \mid T, p \mid f \in c, p \mid \epsilon & (\textit{Program}) \\
\Delta & := & d \mid \textit{opaque}(d) & (\textit{Defintion and scope}) \\
d & := & f\ x_1 \ldots x_n = e \mid f\ x_1 \ldots x_n = \text{ case } e \text{ of } [(\textit{pat}_i, e_i)] & (\textit{Definition}) \\
T & := & \text{data } x = D_1; D_2; \ldots; D_n & (\textit{Data type definition}) \\
\textit{pat} & := & D(x_1, \ldots, x_n) & (\textit{Pattern})
\end{array}$$

In the calculus, every definition is toplevel (λ-lifted) and there isn't any nested case.

For the moment we consider data constructors to be saturated, ie fully applied (hence the special application syntax). That makes defining $\mathcal{D}[\![]\!]$ less painful, because we avoid a quadratic number (in the arity of the data contrustors) of axioms. cf S_4 below, for example.

1.2 FOL

$$\begin{array}{rcll}
t & := & x \mid \text{app}(t_1, t_2) \mid D(t, \ldots, t) \mid f \mid n \mid \texttt{BAD} \mid \texttt{UNR} \mid \text{CF}(t) & (\textit{Term}) \\
\phi & := & \forall x.\phi \mid \phi \to \phi \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \textit{true} \mid t = t \mid \text{CF}(t) & (\textit{Formula})
\end{array}$$

We always give the following equations to define crash-freeness:

$$\text{CF}(\texttt{UNR}), \neg\text{CF}(\texttt{BAD}), \forall f, x\ \text{CF}(f) \wedge \text{CF}(x) \implies \text{CF}(\textit{app}(f, x))$$

Note that it's only the equation we *always* give, but there some other equations that will arise from the translations below. It does not follow exactly the semantics given by the popl paper, but we hope it'll be a sufficient approximation!

1.3 Contracts

$$\begin{array}{rcl}
c & := & x : c_1 \to c_2 \\
& \mid & (c_1, c_2) \\
& \mid & \{x \mid e\} \\
& \mid & \texttt{Any}
\end{array}$$

Semantics of contract satisfaction (exactly the same as in the POPL paper):

$$\begin{array}{rcl}
e \in \{x \mid p\} & \iff & e \text{ diverges or } (e \text{ is crash-free and } p[e/x] \not\to^\star \{\texttt{BAD}, \texttt{UNR}\}) \\
e \in x : t_1 \to t_2 & \iff & \forall e_1 \in t_1, (e\ e_1) \in t_2[e_1/x] \\
e \in (t_1, t_2) & \iff & e \text{ diverges or } (e \to^\star (e_1, e_2) \text{ and } e_1 \in t_1, e_2 \in t_2) \\
e \in \texttt{Any} & \iff & \textit{True}
\end{array}$$

2 Translation

We define several translations: $\mathcal{E}[\![]\!], \mathcal{D}[\![]\!], \mathcal{T}[\![]\!], \mathcal{S}[\![]\!], [\![]\!]$.

$$\begin{array}{rcl}
\mathcal{E}[\![]\!] & :: & \textit{Expression} \to \textit{Term} \\
\mathcal{D}[\![]\!] & :: & \textit{Definition} \to \textit{FOF} \\
\mathcal{T}[\![]\!] & :: & \textit{Data type} \to \{\textit{FOF}\} \\
\mathcal{S}[\![]\!] & :: & \textit{Expression} \to \textit{Contract} \to \textit{FOF} \\
[\![]\!] & :: & \textit{Program} \to \{\textit{FOF}\}
\end{array}$$

2.1 $\mathcal{E}[\![]\!]$

$\mathcal{E}[\![e]\!]$ is a term, the translation is really straightforward.

$$\begin{aligned}
\mathcal{E}[\![x]\!] &= x && (1)\\
\mathcal{E}[\![f]\!] &= f && (2)\\
\mathcal{E}[\![e_1\ e_2]\!] &= app(e_1, e_2) && (3)\\
\mathcal{E}[\![D(e_1,\dots,e_n)]\!] &= D(\mathcal{E}[\![e_1]\!],\dots,\mathcal{E}[\![e_n]\!]) && (4)\\
\mathcal{E}[\![\mathtt{BAD}]\!] &= \mathtt{BAD} && (5)\\
\mathcal{E}[\![n]\!] &= n && (6)\\
& && (7)
\end{aligned}$$

As usual, we use left associativity.

2.2 $\mathcal{T}[\![]\!]$

$\mathcal{T}[\![T]\!]$ is a set of first-order formulae which we break down in four parts: $\mathcal{T}[\![\text{data}\, T = D_1;\dots;D_n]\!] = S_1 \cup S_2 \cup S_3 \cup S_4$.

First, for each D_i of arity n_i we introduce selectors sel_{k,D_i}, which are projections of $D_i(x_1,\dots,x_{ni})$ on its k-th component:

$$S_1 := \{\forall x_1,\dots,x_{n_i}. \bigwedge_{1\le j\le n} sel_{j,D_i}(D_i(x_1,\dots,x_{n_i})) = x_j \mid 1 \le i \le n\}$$

For each pair of different constructors D_i, D_j, we state that they can never map to the same value:

$$S_2 := \{\forall x_1,\dots,x_{n_i}\ \forall y_1,\dots,y_{n_j}.D_i(x_1,\dots,x_{n_i}) \ne D_j(y_1,\dots,y_{n_j}) \mid 1 \le i < j \le n\}$$

Then, we have to give crash-freeness conditions for each D_i:

$$S_3 := \{\forall x_1,\dots,x_{n_i}.(\mathrm{CF}(x_1) \wedge \dots \wedge \mathrm{CF}(x_{n_i}) \leftrightarrow \mathrm{CF}(D_i(x_1,\dots,x_{n_i}))) \mid 1 \le i \le n\}$$

Note that we have an equivalence in the case of a data constructor whereas we only have an (direct) implication for functions. It's because a function, due to laziness, could not use all its arguments. For example, $fst(1, BAD)$ is crash-free but its arguments aren't.

Finally, we have to say that none of the D_i is unreachable:

$$S_4 := \{D_i(x_1,\dots,x_n) = \mathtt{UNR} \to x_1 = \mathtt{UNR} \vee \dots \vee x_n = \mathtt{UNR} \mid 1 \le i \le n\}$$

NB: We first started with $S_4 := \{D_i \ne \mathtt{UNR} \mid 1 \le i \le n\}$, but we couldn't prove that the (add x y), defined with peano integers, satisfied $(x = 0 \to y = 0 \to add\ x\ y = 0)$. It's also bizarre to state that the function was UNR but not its application.

2.3 $\mathcal{D}[\![]\!]$

$\mathcal{D}[\![d]\!]$ is a first-order formula.

$$\begin{aligned}
\mathcal{D}[\![f\ \overline{x} = e]\!] &= \forall x_1 \dots x_n.\mathcal{E}[\![f\ x_1 \dots x_n]\!] = \mathcal{E}[\![e]\!] && (8)\\
\mathcal{D}[\![f\ \overline{x} = \text{case } e \text{ of } [D_i(\overline{z}) \mapsto e_i]]\!] &= \forall x_1 \dots x_n.(\bigwedge_i (\forall \overline{z}\ \mathcal{E}[\![e]\!] = \mathcal{E}[\![D_i(\overline{z})]\!] \to \mathcal{E}[\![f\ \overline{x}]\!] = \mathcal{E}[\![e_i]\!]) && (9)\\
&\quad \wedge \mathcal{E}[\![e]\!] = \mathtt{BAD} \to \mathcal{E}[\![f\ x_1 \dots x_n]\!] = \mathtt{BAD}) && (10)\\
&\quad \wedge((\bigwedge_i e \ne D_i(sel_{1,D_i}(e),\dots,sel_{n_i,D_i}(e)) \wedge \mathcal{E}[\![e]\!] \ne \mathtt{BAD}) && (11)\\
&\qquad \to \mathcal{E}[\![f\ x_1 \dots x_n]\!] = \mathtt{UNR}) && (12)
\end{aligned}$$

An alternative to using selectors would be to write something along the line of $\exists y_1, \ldots, y_{D_i}.e \neq D_i(y_1, \ldots, y_{D_i})$. It is equivalent but the skolemisation process of equinox will anyway turn those existentials in selectors functions. But if we pattern match D_i in two different functions, we would get two different selectors (the same selector with two different names) so it's better (for the efficiency of the proof) to define selectors for each D_i once in for all and use it as much as possible afterwards.

Equation (10) is not restrictive even in the case of laziness because e is always evaluated. (TODO, but not always fully evaluated...)

Equation (11) and (12) state that if we have a type mismatch than the code is UNR. It should never occur because we're assuming that the code has already been typechecked.

2.4 $\mathcal{S}[\![]\!]$

$\mathcal{S}[\![e \in c]\!]$ is a first-order formula.

$$\begin{aligned}
\mathcal{S}[\![e \in \texttt{Any}]\!] &= true && (13)\\
\mathcal{S}[\![e \in \{x \mid u\}]\!] &= e = \texttt{UNR} \vee (\mathrm{CF}(\mathcal{E}[\![e]\!]) \wedge \mathcal{E}[\![u[e/x]]\!] \neq \texttt{BAD} \wedge \mathcal{E}[\![u[e/x]]\!] \neq False) && (14)\\
\mathcal{S}[\![e \in x : c_1 \to c_2]\!] &= \forall x_1.\mathcal{S}[\![x_1 \in c_1]\!] \to \mathcal{S}[\![e\ x_1 \in c_2[x_1/x]]\!] && (15)
\end{aligned}$$

$False$ is a data constructor here.

Remark: we follow the semantics of the POPL paper but it's a bit restrictive. e.g. in equation 14 we could use the alternate semantics (namely B1 in the POPL paper) :

$$\mathcal{S}[\![e \in \{x \mid u\}]\!] = e = \texttt{UNR} \vee (\mathcal{E}[\![u[e/x]]\!] \neq \texttt{BAD} \wedge \mathcal{E}[\![u[e/x]]\!] \neq False)$$

2.5 $[\![]\!]$

$[\![p]\!]$ defines the translation of a program to a theory (a set of FO formulae)

We have to give some semantics to CF(), BAD and UNR, which are given below.

$$\begin{aligned}
[\![\epsilon]\!] &= \{\mathrm{CF}(\texttt{UNR}), \neg\mathrm{CF}(\texttt{BAD}), \forall f, x\ \mathrm{CF}(f) \wedge \mathrm{CF}(x) \implies \mathrm{CF}(app(f, x))\} && (16)\\
[\![d, p]\!] &= \mathcal{D}[\![d]\!] \cup [\![p]\!] && (17)\\
[\![opaque(d), p]\!] &= \mathcal{D}[\![d]\!] \cup [\![p]\!] && (18)\\
[\![T, p]\!] &= \mathcal{T}[\![T]\!] \cup [\![p]\!] && (19)\\
[\![f \in c, p]\!] &= \mathcal{S}[\![f \in c]\!] \cup [\![p]\!] && (20)\\
& && (21)
\end{aligned}$$

3 User interaction

The user provides a program p which consists of functions definition (either opaque or transparent), data types definition and claims that functions satisfies contracts.

Then, for each function definition $d_f := f\ x_1 \ldots x_n = e$ of a function f that has to satisfy the set of contracts C_f, we construct the context $C = p \setminus (d_f \cup C_f)$, which is basically the program p without the definition of f and the contracts it has to satisfy.

We then want to check (with equinox) that:

$$[\![C]\!] \cup \{\mathcal{D}[\![f\ x_1 \ldots_n = e[f^\star/f]]\!], \bigwedge_{c \in C_f} \mathcal{S}[\![f^\star \in c]\!]\} \models \bigwedge_{c \in C_f} \mathcal{S}[\![f \in c]\!]$$

Which we rewrite as:

$$[\![C]\!] \cup \{\mathcal{D}[\![f\ x_1 \ldots_n = e[f^\star/f]]\!], \bigwedge_{c \in C_f} \mathcal{S}[\![f^\star \in c]\!], \bigvee_{c \in C_f} \neg\mathcal{S}[\![f \in c]\!]\} \models \bot$$

4 Example

4.1 Semantics

Consider the following program (the integer after a data constructor is its arity):

```
data List = Nil 0
          | Cons 2;;

data Bool = True 0
          | False 0;;

copy x = case x of
  | Nil -> x
  | Cons a b -> Cons a (copy b);;

copy ::: a:{x : True} -> {y:True};;
```

The contract states that if we give to copy a crash-free argument x, copy will return a crash-free result y. Note that [Just BAD] is not crash-free in our sense, so we can't say anything about it. TODO (or can we?)

4.2 Arguments order

```
data Nat = Zero 0
         | Succ 1;;

data Bool = True 0
          | False 0;;

add x y = case x of
        | Zero -> y
        | Succ z -> Succ (add z y);;

isZero x = case x of
         | Zero -> True
         | Succ z -> False;;

not x = case x of
      | True -> False
      | False -> True;;

add ::: a:{x : True} -> {y:True} -> {z: True};;             -- C1
add ::: a:{x : isZero x} -> {y:isZero y} -> {z: True};;     -- C2
add ::: a:{x : isZero x} -> {y:isZero y} -> {z: isZero z};; -- C3
```

As of today, we can prove C1 and C2 but not C3. Note that we can prove C2 only since the modification of S_4 in $\mathcal{D}[\![]\!]$.

In the definition of *add*, if we replace $Succ(addzy)$ by $Succ(addyz)$, we can't prove anything which isn't very surprising because we're to do the recursion on the first variable, not on the second one. All the more so that they don't have the same contract.

4.3 Multiple contracts - Take 1

We suppose defined datatypes for Nat and Bool, just as in the example above.

```
add a b = case a of
  | Zero -> b
  | Succ x -> Succ (add x b);;

mult a b = case a of
  | Zero -> Zero
  | Succ x -> add b (mult x b);;

notZero x = case x of
  | Zero -> False
  | Succ a -> True;;

notZero ::: a:{x:True} -> {y: True};;
add  ::: a:{x: True} -> b:{y: True} -> {z: True};;
add  ::: a:{x: notZero x} -> b:{y: True} -> {z: True};;
mult ::: a:{x: notZero x} -> b:{y: notZero y} -> {z: True};;
```

Equinox diverges when we try to check mult's contract.

The reason stems from that *mult* (*Succ x*) *b* calls recursively *mult x b* The induction hypothesis says that if a is nonzero and b is nonzero then the result is `Ok`. But in this case x may be zero, so the induction hypothesis does not apply immediately. Now, the theorem prover could do a split on x (either Zero or Succ_) and in one case the result should follow from the induction hypothesis, but in the other case we are stuck! Because we don't actually know that $multp 0 b$ is `Ok` (*multp* is the recursive version of the function, whose contract we assumed but no further defining equations!)

A possible fix, which is only possible when the function is total is to write

$$f ::: a_1 : \{x \mid p_1(x)\} \to \cdots \to a_n\{x \mid p_n(x)\}$$

as

$$f ::: \texttt{Ok} \to \cdots \to \texttt{Ok} \to \{x \mid if\ (p_i(a_i))\ then\ p_n(x)\ else\ True)$$