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

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

For the moment we consider data constructors to be saturated, ie fully applied (hence the special application syntax).

1.2 FOL

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

$\text{CF}(t)$ holds iff t satisfies `Ok`.

1.3 Contracts

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

Semantics of contract satisfaction:

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

2 Translation

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

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

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

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

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

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

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

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 composant, so that we can express that constructors are injective :

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

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

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

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

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

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

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

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

An alternative to using selectors would be to write something along the line of $\exists y_1, \dots, y_{D_i}.e \neq D_i(y_1, \dots, y_{D_i})$. 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.

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

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

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

$False$ is a data constructor here.

Remark: we follow the semantics of the POPL paper but it's a bit restrictive. e.g. in equation 13 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)

$$\begin{aligned}
[\![\epsilon]\!] &= \emptyset && (16)\\
[\![d, p']\!] &= \mathcal{D}[\![d]\!] \cup [\![p]\!] && (17)\\
[\![opaque(d), p]\!] &= \mathcal{D}[\![d]\!] \cup p && (18)\\
[\![T, p]\!] &= \mathcal{T}[\![T]\!] \cup [\![p]\!] && (19)\\
[\![f \in c, p]\!] &= \mathcal{S}[\![f \in c]\!] \cup [\![p]\!] && (20)\\
& && (21)
\end{aligned}$$

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 Remarks

- What to do with mutually recursive functions?
- Explore the variants translations (cf variants of the popl paper)