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 $\mathcal{H}$: Haskell subset

Our source language $\mathcal{H}$ is defined below. Note that λ abstractions and case expressions can appear anywhere in the code.

assert (e, c) in e' means that if e does not satisfy the contract c then the compiler should return and otherwise execute e'.

e 'satisfies' c is a boolean expression which is True iff the expression e satisfies the contract c in the current context.

$$\begin{array}{rcll}
u, e & := & x \mid f \mid \lambda x.\ e \mid e\ e \mid D(e, \ldots, e) \mid \texttt{BAD} \mid n \mid e \text{ `satisfies` } c \mid \text{assert } (e, c) \text{ in } e & (\mathit{Expression}) \\
 & \mid & \text{case } e \text{ of } [(p_i, e_i)] \mid \text{CF}(e) & \\
p & := & \Delta, p \mid T, p \mid f \in c, p \mid \epsilon & (\mathit{Program}) \\
\Delta & := & d \mid \mathit{opaque}(d) & (\mathit{Defintion\ and\ scope}) \\
d & := & f\ x_1 \ldots x_n ::: c = e \mid f\ x_1 \ldots x_n ::: c = \text{ case } e \text{ of } [(\mathit{pat}_i, e_i)] & (\mathit{Definition}) \\
T & := & \text{data } x = \mathit{DCons} & (\mathit{Data\ type\ definition}) \\
\mathit{DCons} & := & \epsilon \mid D; \mathit{DCons} \mid D ::: c; \mathit{DCons} & (\mathit{List\ of\ data\ constructors}) \\
\mathit{pat} & := & D(x_1, \ldots, x_n) & (\mathit{Pattern})
\end{array}$$

1.2 $\mathcal{H}'$: λ-lifted haskell subset

This language is as expressive as $\mathcal{H}$ but contains fewer constructs. Compared to $\mathcal{H}$ the restrictions are:

- No λ, ie every λ is lifted to a function definition at the toplevel
- No case expressions which are lifted out to the toplevel (ie pattern matching is only possible for a function declaration)
- No *assert* which are syntactic sugar for case expressions

From now on, expressions will always refer to $\mathcal{H}'$ expressions.

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

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

1.3 From $\mathcal{H}$ to $\mathcal{H}'$

1.3.1 Step 1: *assert*

We translate every expression $e = \text{assert } (e_c, c) \text{ in } e'$ in case $e_c`satisfies`c$ of $True- > e'|False- > \texttt{BAD}$.

1.3.2 Step 2: $\lambda - lifting$

GHC's API can do that.

1.3.3 Step 3: *case* lifting

...

1.4 FOL

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

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

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

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!

1.5 Contracts

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

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

$$\begin{array}{rcl} e \in \{x \mid p\} & \iff & e \text{ diverges or } p[e/x] \not\to^{\star} \{\texttt{BAD}, False\} \\ e \in x : t_1 \to t_2 & \iff & \forall e_1 \in t_1, (e\ e_1) \in t_2[e_1/x] \\ e \in (t_1, t_2) & \iff & e \text{ diverges or } (e \to^{\star} (e_1, e_2) \text{ and } e_1 \in t_1, e_2 \in t_2) \\ e \in c_1 \wedge c_2 & \iff & e \in c_1 \text{ and } e \in c_2 \\ e \in c_1 \vee c_2 & \iff & e \in c_1 \text{ or } e \in c_2 \end{array}$$

2 CF-ness and unreachability

2.1 CF-ness

In the POPL paper, CF was introduced in the contract satisfaction semantics because it was required in order to define the wrapping. Given that we do not use any wrapping, we can make a separation between crash-freeness and predicate satisfaction now, as shown in the contracts semantics, with the special syntax.

It leads to some tricky cases:

Assume a reasonable definition of *add*. Then, this contract does not hold.

$$add ::: a : \{x \mid True\} \to \{y \mid True\} \to \{z \mid z \geq a\}$$

Why? Because now that we have removed crash-freeness, we can have x and y instantiated to `BAD`. And then, $add\ x\ y \geq z$ becomes `BAD`. Or the semantics say that the predicate should not evaluate to `BAD`.

Now, the correct contract is:

$$add ::: a : \{!x \mid cf(x)\} \to \{!y \mid cf(y)\} \to \{!z \mid z \geq a\}$$

3 Translation

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

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

3.1 $\mathcal{E}[\![]\!]$

$\mathcal{E}[\![e]\!]$ is a pair (term,formula), the only tricky translations is for satisfies. We write $\mathcal{E}_t[\![]\!]$ and $\mathcal{E}_f[\![]\!]$ the projection of $\mathcal{E}[\![]\!]$ on the first and second component.

$$\begin{array}{rcll} \mathcal{E}[\![x]\!] & = & (x, \emptyset) & (1) \\ \mathcal{E}[\![f]\!] & = & (f, \emptyset) & (2) \\ \mathcal{E}[\![e_1\ e_2]\!] & = & (app(\mathcal{E}_t[\![e_1]\!], \mathcal{E}_t[\![e_2]\!]), \mathcal{E}_f[\![e_1]\!] \cup \mathcal{E}_f[\![e_2]\!]) & (3) \\ \mathcal{E}[\![D(e_1, \dots, e_n)]\!] & = & (D(\mathcal{E}_t[\![e_1]\!], \dots, \mathcal{E}_t[\![e_n]\!]), \cup_{1 \le i \le n} \mathcal{E}_f[\![e_i]\!]) & (4) \\ \mathcal{E}[\![\mathtt{BAD}]\!] & = & (\mathtt{BAD}, \emptyset) & (5) \\ \mathcal{E}[\![n]\!] & = & (n, \emptyset) & (6) \\ \mathcal{E}[\![e \text{ `satisfies` } c]\!] & = & (\mathcal{E}_t[\![e]\!], \mathcal{E}_f[\![e]\!] \cup \forall x. \mathcal{S}[\![x \in c]\!] \leftrightarrow x \text{ `}satisfies\text{` } c = True) & (7) \\ \mathcal{E}[\![\mathrm{CF}(e)]\!] & = & \mathrm{CF}(e) & (8) \\ & & & (9) \end{array}$$

TODO: take car of the FV in c for satisfies. As usual, we use left associativity.

3.2 $\mathcal{T}[\![]\!]$

$\mathcal{T}[\![T]\!]$ is a set of first-order formulae which we break down in five parts: $\mathcal{T}[\![\text{data } T = D_1; \dots; D_n]\!] = S_1 \cup S_2 \cup S_3 \cup S_4 \cup S_5$.

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:

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

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.

We also 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 $S_4 := \{D_i \neq \texttt{UNR} \mid 1 \leq i \leq n\}$, 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.

Finally, we give contracts to each data constructors. It is the contract c specified by the user. If no contract is given for a specific data constructor, then we assume that the contract is $\texttt{Ok} \to \cdots \to \texttt{Ok}$. We write the contract c as $c_1 \to \cdots \to c_{n_i}$

$$S_5 := \{\forall x_1, \ldots, x_{n_i} \mathcal{S}[\![D_i(x_1, \ldots, x_{n_i}) \in c]\!] \leftrightarrow \bigwedge_{1 \leq i \leq n_i} \mathcal{S}[\![x_i \in c_i]\!] \mid 1 \leq i \leq n\}$$

3.3 $\mathcal{D}[\![]\!]$

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

$$\begin{aligned}
\mathcal{D}[\![f\ \overline{x} = e]\!] &= \forall x_1 \ldots x_n.\mathcal{E}_t[\![f\ x_1 \ldots x_n]\!] = \mathcal{E}_t[\![e]\!] \cup \mathcal{E}_f[\![e]\!] && (10)\\
\mathcal{D}[\![f\ \overline{x} = \text{case } e \text{ of } [D_i(\overline{z}) \mapsto e_i]]\!] &= \forall x_1 \ldots x_n.(\bigwedge_i (\forall \overline{z}\ \mathcal{E}_t[\![e]\!] = \mathcal{E}_t[\![D_i(\overline{z})]\!] \to \mathcal{E}_t[\![f\ \overline{x}]\!] = \mathcal{E}_t[\![e_i]\!]) && (11)\\
&\quad \wedge \mathcal{E}_t[\![e]\!] = \texttt{BAD} \to \mathcal{E}_t[\![f\ x_1 \ldots x_n]\!] = \texttt{BAD}) && (12)\\
&\quad \wedge ((\bigwedge_i e \neq D_i(sel_{1,D_i}(e), \ldots, sel_{n_i,D_i}(e)) \wedge \mathcal{E}_t[\![e]\!] \neq \texttt{BAD}) && (13)\\
&\qquad \to \mathcal{E}_t[\![f\ x_1 \ldots x_n]\!] = \texttt{UNR}) && (14)\\
&\quad \cup \mathcal{E}_f[\![e]\!] && (15)
\end{aligned}$$

Note that in those equations, there is no $\mathcal{E}_f[\![f\ x_1 \ldots x_n]\!]$ because it is always the empty set.

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.

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.

3.4 $\mathcal{S}[\![]\!]$

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

$$\begin{aligned}
\mathcal{S}[\![e \in \{x \mid u\}]\!] &= e = \texttt{UNR} \vee (\mathcal{E}_t[\![u[e/x]]\!] \neq \texttt{BAD} \wedge \mathcal{E}_t[\![u[e/x]]\!] \neq False) && (16)\\
&\quad \cup \mathcal{E}_f[\![e]\!] \cup \mathcal{E}_f[\![u[e/x]]\!] && (17)\\
\mathcal{S}[\![e \in x : c_1 \to c_2]\!] &= \forall x_1.\mathcal{S}[\![x_1 \in c_1]\!] \to \mathcal{S}[\![e\ x_1 \in c_2[x_1/x]]\!] && (18)\\
\mathcal{S}[\![e \in c_1 \wedge c_2]\!] &= \mathcal{S}[\![e \in c_1]\!] \wedge \mathcal{S}[\![e \in c_2]\!] && (19)\\
\mathcal{S}[\![e \in c_1 \vee c_2]\!] &= \mathcal{S}[\![e \in c_1]\!] \vee \mathcal{S}[\![e \in c_2]\!] && (20)\\
& && (21)
\end{aligned}$$

$False$ 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) :

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

3.5 $[\![]\!]$

$[\![p]\!]$ 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.

$$\begin{aligned}
[\![\epsilon]\!] &= \{\mathrm{CF}(\mathtt{UNR}), \neg\mathrm{CF}(\mathtt{BAD}), \forall f, x\ \mathrm{CF}(f) \wedge \mathrm{CF}(x) \implies \mathrm{CF}(app(f,x))\} && (22)\\
[\![d,p]\!] &= \mathcal{D}[\![d]\!] \cup [\![p]\!] && (23)\\
[\![opaque(d),p]\!] &= \mathcal{D}[\![d]\!] \cup [\![p]\!] && (24)\\
[\![T,p]\!] &= \mathcal{T}[\![T]\!] \cup [\![p]\!] && (25)\\
[\![f \in c, p]\!] &= \mathcal{S}[\![f \in c]\!] \cup [\![p]\!] && (26)\\
& && (27)
\end{aligned}$$

4 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 $d_f := f\ x_1 \dots x_n = e$ 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:

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

Which we rewrite as:

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

4.1 Checking the contracts in the right order

Here is a little example showing that we should be careful about we belong to a theory.

Assume that we have a module that contains two functions defintion f and g and two contracts : $f ::: Any \to Ok$ and $g ::: Any \to Ok$. We assume that those contract do not hold, for example if f is *not* and g is *id*.

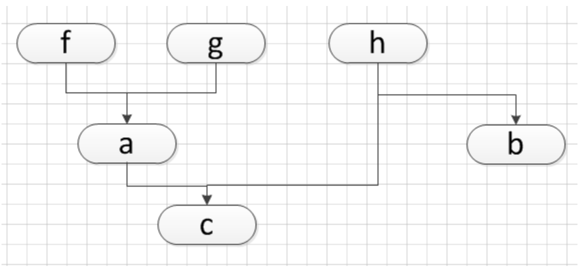
Now, we want to check f's contract. We take out f's contract, translate the function definition for f and g and add the contract satisfation formulae for g's contract.

But, given that g's contract does not hold, we can derive $\bot$ and then prove that f's contract hold.

Now the user is happy, and does the same thing for g's contract. It holds too, given that we can also derive $\bot$ for f's contract.

Finally, the user is really happy, but in fact he has proven nothing.

The solution is to only include the formulae that belongs to a functions that are actually used. For example, to prove a's contract, we should only include f and g formulae.



5 Example

5.1 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.

5.2 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 : isZero x} -> {!y:isZero y} -> {!z: isZero z};;
```

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.

A lemma stating (+) commutativity would be helpful here.

5.3 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

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

And know we can prove *mult*'s contract, but it's not very satisfying.

A stupid fix was to provide defining equations for *multp* (the same as for *mult*). Indeed, we could prove that *mult* satisfied its contract, but it was trivially true...

Current idea is to unfold the definition a fixed number of times.

6 De-appification and removed CFness

6.1 De-appification

Instead of always using the *app* function when doing function application, we now use directly the function name if it is fully applied. For example, if f is of arity 2, fxy is now translated to $f(x, y)$ instead of $app(app(f, x), y)$. To deal with partially applied functions, we can use *f_ptr* and some extra axioms.

This transformation makes things much faster for equinox, especially given that in most cases, function are totally applied.