Contracts for Haskell

Simon (PJ), Dimitrios, Koen, Charles-Pierre

Goals

We would like to write things like:

```
head :::  { x | not (null x) } -> Ok
head xs = case xs of
| [] -> BAD
| x:_ -> x
```

```
map :::  (Ok -> Ok) -> xs:Ok ->
             { ys | length ys = length xs }
```

Note: `Ok` is just syntactic sugar for $\{x \mid \mathit{True}\}$

Contract syntax

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

- Predicates should be any boolean Haskell expression.
- Everything (even errors!) satisfies `Any`

  ```
  fst ::: (Ok, Any) -> Ok
  ```
- Named arguments: x is any value satisfying c_1.

Crash-freeness

Morally, an expression e is crash-free (cf) iff any `BAD`-free context in which we execute the expression in does not yield `BAD`.

Laziness makes things tricky:

- `Just BAD`: not CF
- `(3,error ''foo'')`: not CF
- `fst (3,error ''foo'')`: CF

And now to the operational semantics of contracts!

Contract semantics

$$
\begin{array}{rcl}
e ::: \{x \mid p\} & \Longleftrightarrow & e \text{ diverges or} \\
 & & (e \text{ is crash-free and } p[e/x] \not\rightarrow^{\star} \{\texttt{BAD}, \mathit{False}\}) \\
e ::: x : t_1 \rightarrow t_2 & \Longleftrightarrow & \text{for all } e_1, e_1 ::: t_1 \text{ implies } (e\ e_1) ::: t_2[e_1/x] \\
e ::: (t_1, t_2) & \Longleftrightarrow & e \text{ diverges or} \\
 & & (e \rightarrow^{\star} (e_1, e_2) \text{ and } e_1 ::: t_1 \text{ and } e_2 ::: t_2) \\
e ::: \texttt{Any} & \Longleftrightarrow & \mathit{True}
\end{array}
$$

Contract checking

Two roads: runtime checking and static checking.

- Runtime: Findler & Felleisen, many others ...
- The road less taken: static checking

Our simple plan:

Haskell + Contracts → First-order logic → Automated Theorem Prover (equinox)

Equinox is Koen's automated theorem prover. It's a FO theorem prover with equality and is under active development.

Our language

Ok, not really Haskell. Wlog we make these asumptions:

- Top-level only pattern-matching
- λ-lifting (top-level function definitions)
- Saturated data constructors (l.o.g. here but only for the sake of clarity)

From now on, we only consider well-typed expressions.

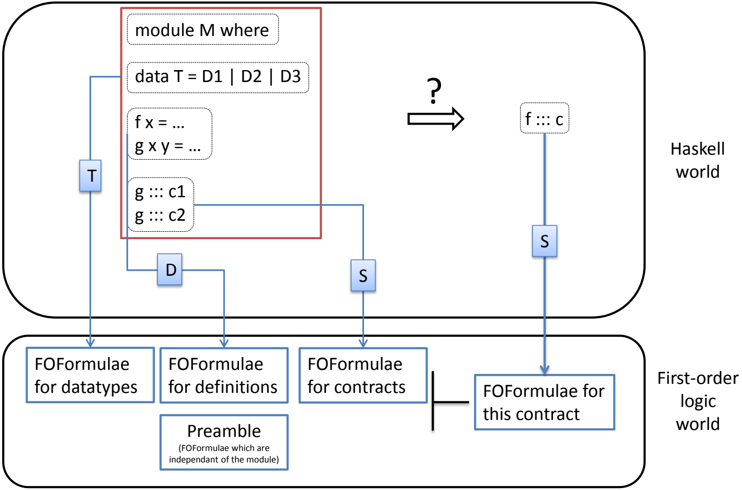
Our syntax

$$
\begin{array}{rcl}
u, e & := & x \mid f \mid e\ e \mid D(e, \ldots, e) \mid \mathtt{BAD} \\
mod & := & def, mod \mid tdecl, mod \mid f ::: c, mod \mid \epsilon \\
def & := & f\ x_1 \ldots x_n = e \mid f\ x_1 \ldots x_n = \ \text{case}\ e\ \text{of}\ [(pat_i, e_i)] \\
pat & := & D(x_1, \ldots, x_n) \\
tdecl & := & \text{data}\ \ T = dcons \\
dcons & := & \epsilon \mid D; dcons \mid D ::: c; dcons
\end{array}
$$

The big picture

Recall that $mod := def, mod \mid T, mod \mid f ::: c, mod \mid \epsilon$; a module is a list of function definitions, data definitions and claims that functions satisfy their contracts.

The big picture



What we need

Theorem (unproven)

$T_p \vdash T_c \implies f ::: c$

Unfortunately, so far we're only believers.

Given this theorem we can use Koen's theorem prover to decide if $T_p \vdash T_c$.

Expressions become first-order logic terms

$$
\begin{aligned}
\mathcal{E}[\![x]\!] &= x \\
\mathcal{E}[\![f]\!] &= f \\
\mathcal{E}[\![e_1\ \ e_2]\!] &= app(\mathcal{E}[\![e_1]\!], \mathcal{E}[\![e_2]\!]) \\
\mathcal{E}[\![D(e_1, \ldots, e_n)]\!] &= D(\mathcal{E}[\![e_1]\!], \ldots, \mathcal{E}[\![e_n]\!]) \\
\mathcal{E}[\![\mathtt{BAD}]\!] &= \mathtt{BAD}
\end{aligned}
$$

- Looks reasonable... But what's this "app" ?
- Why not $\mathcal{E}[\![e_1\ \ e_2]\!] = e_1(e_2)$ with e_1 being a FO function?
- To keep things first-order...
- Because we will need to quantify over an arbitrary expression (that includes functions)

Formulae in the preamble

We gave an (informal) definition of crash-freeness with the semantics of contract satisfaction. But here, we use CF as a logic predicate.
We always add to our theory the following formulae, only approximating the intended semantics:

$$\forall f, x.\ \mathrm{CF}(f) \wedge \mathrm{CF}(x) \implies \mathrm{CF}(app(f, x))$$

$$\neg\mathrm{CF}(\mathtt{BAD})$$

Translating modules

$$
\begin{aligned}
\mathcal{E}[\![]\!] &:: \mathit{Expression} \rightarrow \mathit{Term} \\
\mathcal{D}[\![]\!] &:: \mathit{Definition} \rightarrow \{\mathit{FOF}\} \\
\mathcal{T}[\![]\!] &:: \mathit{Data\ type\ declaration} \rightarrow \{\mathit{FOF}\} \\
\mathcal{S}[\![]\!] &:: \mathit{Expression} \rightarrow \mathit{Contract} \rightarrow \{\mathit{FOF}\}
\end{aligned}
$$

$\mathcal{D}[\![]\!]$ – Definition translation (easy case)

Let f be defined as $f\ x = (x, x)$. It gives rise to the formula $\forall x.\ app(f, x) = \mathrm{Pair}(x, x)$.

- In the formula f is an uninterpreted constant symbol.
- Pair is a function in FO logic.
  We will give formulae for it soon!

In general:

$$\mathcal{D}[\![f\ x\ y = e]\!] = \forall x\ y.\ \mathcal{E}[\![f\ x\ y]\!] = \mathcal{E}[\![e]\!]$$

$\mathcal{T}[\![]\!]$ – Data declaration translation

$\mathcal{T}[\![data T = D_1 \mid \cdots \mid D_n]\!] =?$

Reasonable requirements:

- Injectivity
- CF conditions

Let's give some details...

$\mathcal{T}[\![]\!]$ – Data constructor injectivity

$\mathcal{T}[\![data T = D_1 \mid D_2]\!] =?$
Assume the following arities: D_1:2, D_2:1.

$$\forall x_1, x_2.\ \forall y.\ D_1(x_1, x_2) \neq D_2(y)$$

Do this for every pair of constructors!

$\mathcal{T}[\![]\!]$ – Data constructor crash-freeness

$\mathcal{T}[\![data T = D_1 \mid D_2]\!] =?$

$$\forall x_1, x_2.\ \mathrm{CF}(x_1) \wedge \mathrm{CF}(x_2) \leftrightarrow \mathrm{CF}(D_1(x_1, x_2))$$
$$\forall y.\ \mathrm{CF}(y) \leftrightarrow \mathrm{CF}(D_2(y))$$

Values of type T are crash-free iff they are created with crash-free values.

- $\rightarrow$ also true for cf function application
- $\leftarrow$ not true even for cf functions
  (recall that `fst (3,BAD)` is crash-free)

$\mathcal{D}[\![]\!]$ – The case of case

```
head xs = case xs of
  | Nil -> BAD
  | Cons(y,ys) -> y
```

translates to:

$$
\begin{aligned}
& app(head, Nil) = \mathtt{BAD} \\
\wedge \quad & \forall y, ys.\ app(head, Cons(y, ys)) = y \\
\wedge \quad & app(head, \mathtt{BAD}) = \mathtt{BAD} \\
\wedge \quad & \forall xs.\ xs \neq Nil \wedge (\forall a\ b.\ xs \neq Cons(a, b)) \\
& \qquad \wedge xs \neq \mathtt{BAD} \implies app(head, xs) = \mathtt{UNR}
\end{aligned}
$$

What is $\mathtt{UNR}$? What is it for?

$\mathcal{S}[\![]\!]$ – Contract satisfaction and the role of $\mathtt{UNR}$

How do we translate that $e ::: c$?

$$
\begin{aligned}
\mathcal{S}[\![e \in \mathtt{Any}]\!] &= \mathit{true} \\
\mathcal{S}[\![e \in \{x \mid u\}]\!] &= e = \mathtt{UNR} \lor (\mathrm{CF}(\mathcal{E}[\![e]\!]) \land \mathcal{E}[\![u[e/x]]\!] \neq \mathtt{BAD} \\
&\qquad \land \mathcal{E}[\![u[e/x]]\!] \neq \mathit{False}) \\
\mathcal{S}[\![e \in x : c_1 \to c_2]\!] &= \forall x_1 . \mathcal{S}[\![x_1 \in c_1]\!] \to \mathcal{S}[\![e\ x_1 \in c_2[x_1/x]]\!]
\end{aligned}
$$

$\mathtt{UNR}$

Used to represent divergence or things that should not happen.

$$\mathit{fst} ::: \mathtt{Ok} \to \mathtt{Ok} \text{ and } 3 ::: \mathtt{Ok}$$

Thus, *fst* 3 should satisfy $\mathtt{Ok}$!
But it cannot happen (it's ill-typed), so we map *fst* 3 to a special value which satisfies $\mathtt{Ok}$. (FOL is untyped!)

$\mathcal{D}[\![]\!]$ – Recursive functions: Take 1

Assume that we want to prove the claim *add* ::: *c* where $c = \mathtt{Ok} \rightarrow \mathtt{Ok} \rightarrow \mathtt{Ok}$.

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

Naively, we would do as usual: ask the theorem prover if

$$\mathcal{D}[\![add = e]\!] \vdash \mathcal{S}[\![add, c]\!]$$

But *add* is recursive, so we would not go very far!
Fix: define a function *add_rec* that we use as IH. We now ask the theorem prover:

$$\mathcal{D}[\![add = e[add/add\_rec]]\!], \mathcal{S}[\![add\_rec, c]\!] \vdash \mathcal{S}[\![add, c]\!]$$

$\mathcal{D}[\![]\!]$ – Recursive functions: Take 1

That yields the following formula:

$$
\begin{array}{ll}
\forall x\ y. & app(app(add, Zero), y) = y \\
\wedge & \forall a.app(app(add, Succ(a)), y) = Succ(app(app(add\_rec, a), y)) \\
\wedge & app(app(add, \mathtt{BAD}), y) = \mathtt{BAD} \\
\wedge & x \neq Zero \wedge (\forall c.\ x \neq Succ(c)) \\
& \quad \wedge\, x \neq \mathtt{BAD} \implies app(app(add, x), y) = \mathtt{UNR}
\end{array}
$$

How about: $app(app(add, x), \mathtt{BAD}) = \mathtt{BAD}$?

$\mathcal{D}[\![]\!]$ – Recursive functions: Take 2

A more complicated example: $mult ::: c$ where

$$c = \{x \mid x \neq 0\} \to \{y \mid y \neq 0\} \to \{z \mid z \neq 0\}$$

```
mult x y = case x of
  | Zero -> Zero
  | Succ a -> add (mult a y) y
```

$$\mathcal{D}[\![mult = e[mult/mult\_rec]]\!], \mathcal{S}[\![mult\_rec, c]\!] \not\vdash \mathcal{S}[\![mult, c]\!]$$

$\mathcal{D}[\![]\!]$ – Recursive functions: Take 2

```
mult x y = case x of
  | Zero -> Zero
  | Succ a -> add (mult a y) y
```

$$\mathcal{D}[\![mult = e[mult/mult\_rec]]\!], \mathcal{S}[\![mult\_rec, c]\!] \nvdash \mathcal{S}[\![mult, c]\!]$$

Why? Because the IH is too restrictive!

Fix:

$$\begin{array}{rl} \mathrm{D}\ [\![mult = e[mult/mult\_rec]]\!] & \\ \mathrm{S}\ [\![mult\_rec, c]\!] & \vdash \quad \mathcal{S}[\![mult, c]\!] \\ \mathrm{D}\ [\![mult\_rec = e[mult/mult\_rec]]\!] & \end{array}$$

Questions

- *Proof that* $T_p \vdash T_c \implies f ::: c$
- Speed-up the theorem prover (by giving strategies)
- The user should be able to specify lemmas. How?
- Hoare Logic view of contracts (partial spec) vs. liquid types (complete spec)
- Contracts on datatypes
- Not all contracts expressible as Haskell predicates. Need special operator: `‘satisfies‘ c`?