Static Haskell Contract Checking

Dan Rosén

Dimitrios Vytiniotis, Koen Claessen, Simon Peyton Jones

Microsoft Research

September 5, 2012

Contracts

Express correctness of Haskell programs with *contracts*.

$$
\begin{array}{rcll}
C & ::= & (x : C) \to C & \text{dependent function space} \\
  & | & \{x \mid p\} & \text{predicates} \\
  & | & \mathsf{CF} & \text{crash free} \\
  & | & C \& C & \text{conjunction}
\end{array}
$$

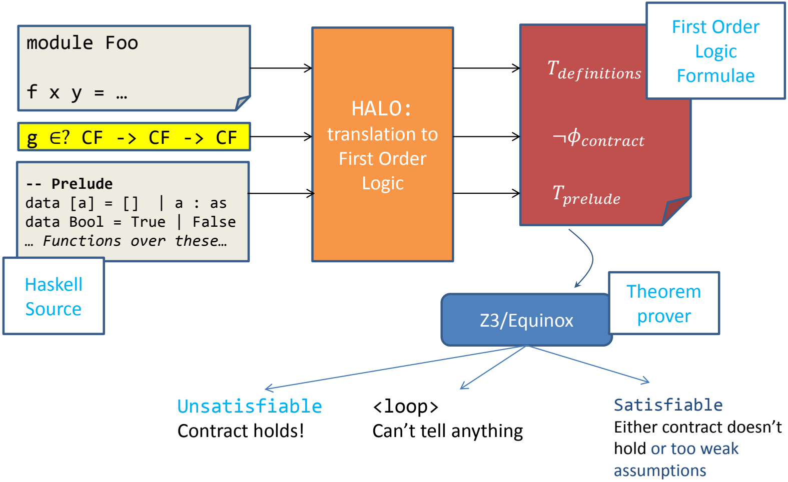
Predicates declared with Haskell functions with ordinary semantics.
Examples:

$$
\begin{array}{l}
\texttt{head} \in \mathsf{CF} \to \mathsf{CF} \\
\texttt{head} \in \{\texttt{xs} \mid \texttt{not (null xs)}\} \to \mathsf{CF} \\
\texttt{head} \in \mathsf{CF}\&\{\texttt{xs} \mid \texttt{not (null xs)}\} \to \mathsf{CF} \\
\texttt{filter} \in (p : \mathsf{CF} \to \mathsf{CF}) \to \mathsf{CF} \to \mathsf{CF}\&\{ys \mid \texttt{all}\ p\ ys\}
\end{array}
$$

Motivation

- Related work: Xu using wrapping, recent work for OCaml
- Interesting aspects of Haskell:
  - lazy/infinite data structures,
  - higher-order,
  - pure

Overview


module Foo
f x y = …
g ∈? CF -> CF -> CF
-- Prelude
data [a] = [] | a : as
data Bool = True | False
… Functions over these…
Haskell Source
HALO: translation to First Order Logic
$T_{definitions}$
$\neg\phi_{contract}$
$T_{prelude}$
First Order Logic Formulae
Z3/Equinox
Theorem prover
Unsatisfiable
Contract holds!
<loop>
Can't tell anything
Satisfiable
Either contract doesn't hold or too weak assumptions

Denotational semantics

Idea: translate a denotational model to FOL.

- Discrimination axioms

$$\mathsf{cons}(x, xs) \neq \mathsf{nil} \neq \mathtt{UNR} \neq \mathtt{BAD}$$

- Injectivity axioms

$$\begin{aligned} \mathsf{cons}_0(\mathsf{cons}(x, xs)) &= x, \\ \mathsf{cons}_1(\mathsf{cons}(x, xs)) &= xs \end{aligned}$$

Using cons_0 we have $\mathsf{cons}(x, xs) = \mathsf{cons}(y, ys) \rightarrow x = y$.

Guiding principle for translation to FOL

Theorem

Assume that $\Sigma \vdash P$ and e_1 and e_2 contain no free term variables. The following are true:

- $[\![e_1]\!] = [\![e_2]\!]$ *iff* $\mathcal{I}(\mathcal{E}\{\!\{e_1\}\!\}) = \mathcal{I}(\mathcal{E}\{\!\{e_2\}\!\})$.
- *If* $\mathcal{T} \wedge \mathcal{P}\{\!\{P\}\!\} \vdash \mathcal{E}\{\!\{e_1\}\!\} = \mathcal{E}\{\!\{e_2\}\!\}$ *then* $[\![e_1]\!] = [\![e_2]\!]$.

$$
\begin{array}{rcl}
[\![x]\!] & = & \rho(x) \\
[\![f]\!] & = & \sigma(f) \\
[\![K\ (\overline{e})]\!] & = & \mathsf{K}(\overline{[\![e]\!]}) \\
[\![e_1\ e_2]\!] & = & \mathsf{app}([\![e_1]\!], [\![e_2]\!]) \\
[\![\mathtt{BAD}]\!] & = & \mathsf{Bad}
\end{array}
\qquad
\begin{array}{rcl}
\mathcal{E}\{\!\{x\}\!\} & = & x \\
\mathcal{E}\{\!\{f\}\!\} & = & f_{ptr} \\
\mathcal{E}\{\!\{K(\overline{e})\}\!\} & = & K(\overline{\mathcal{E}\{\!\{e\}\!\}}) \\
\mathcal{E}\{\!\{e_1\ e_2\}\!\} & = & app(\mathcal{E}\{\!\{e_1\}\!\}, \mathcal{E}\{\!\{e_2\}\!\}) \\
\mathcal{E}\{\!\{\mathtt{BAD}\}\!\} & = & \mathtt{BAD}
\end{array}
$$

Translating Functions to FOL

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

$$
\begin{array}{lcl}
\mathsf{ap}(\mathsf{ap}(\mathsf{map}, f), \mathsf{nil}) & = & \mathsf{nil}, \\
\mathsf{ap}(\mathsf{ap}(\mathsf{map}, f), \mathsf{cons}(x, xs)) & = & \mathsf{cons}(\mathsf{ap}(f, xs), \mathsf{ap}(\mathsf{ap}(\mathsf{map}, f), xs)) \\
\mathsf{ap}(\mathsf{ap}(\mathsf{map}, f), \mathtt{BAD}) & = & \mathtt{BAD}, \\
\mathsf{ap}(\mathsf{ap}(\mathsf{map}, f), x) & = & \mathtt{UNR} \\
\quad \vee\ (\exists\ y\ ys. x = \mathsf{cons}(y, ys)) & & \\
\quad \vee\ x = \mathsf{nil} & & \\
\quad \vee\ x = \mathtt{BAD} & &
\end{array}
$$

Translating Functions to FOL

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

$$
\begin{array}{lcl}
\mathsf{ap}(\mathsf{ap}(\mathsf{map}, f), \mathsf{nil}) & = & \mathsf{nil}, \\
\mathsf{ap}(\mathsf{ap}(\mathsf{map}, f), \mathsf{cons}(x, xs)) & = & \mathsf{cons}(\mathsf{ap}(f, xs), \mathsf{ap}(\mathsf{ap}(\mathsf{map}, f), xs)) \\
\mathsf{ap}(\mathsf{ap}(\mathsf{map}, f), \mathtt{BAD}) & = & \mathtt{BAD}, \\
\mathsf{ap}(\mathsf{ap}(\mathsf{map}, f), x) & = & \mathtt{UNR} \\
\quad \vee\ (\exists\ y\ ys. x = \mathsf{cons}(y, ys)) & & \\
\quad \vee\ x = \mathsf{nil} & & \\
\quad \vee\ x = \mathtt{BAD} & &
\end{array}
$$

$$
\mathsf{ap}(\mathsf{ap}(\mathsf{map}, f), xs) = \mathsf{map}(f, xs)
$$

Translating Functions to FOL

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

$$
\begin{array}{lcl}
\mathsf{map}(f, \mathsf{nil}) & = & \mathsf{nil}, \\
\mathsf{map}(f, \mathsf{cons}(x, xs)) & = & \mathsf{cons}(\mathsf{ap}(f, xs), \mathsf{map}(f, xs)) \\
\mathsf{map}(f, \mathtt{BAD}) & = & \mathtt{BAD}, \\
\mathsf{map}(f, x) & = & \mathtt{UNR} \\
\quad \vee\ (\exists\, y\ ys . x = \mathsf{cons}(y, ys)) & & \\
\quad \vee\ x = \mathsf{nil} & & \\
\quad \vee\ x = \mathtt{BAD} & &
\end{array}
$$

$$
\mathsf{ap}(\mathsf{ap}(\mathsf{map}, f), xs) = \mathsf{map}(f, xs)
$$

Translating Functions to FOL

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

$$
\begin{array}{lcl}
\mathsf{map}(f, \mathsf{nil}) & = & \mathsf{nil}, \\
\mathsf{map}(f, \mathsf{cons}(x, xs)) & = & \mathsf{cons}(\mathsf{ap}(f, xs), \mathsf{map}(f, xs)) \\
\mathsf{map}(f, \mathtt{BAD}) & = & \mathtt{BAD}, \\
\mathsf{map}(f, x) & = & \mathtt{UNR} \\
\quad \vee\ (\exists\ y\ ys . x = \mathsf{cons}(y, ys)) & & \\
\quad \vee\ x = \mathsf{nil} & & \\
\quad \vee\ x = \mathtt{BAD} & &
\end{array}
$$

$$
\mathsf{ap}(\mathsf{ap}(\mathsf{map}_{\mathsf{ptr}}, f), xs) = \mathsf{map}(f, xs)
$$

Translating Functions to FOL

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

$$
\begin{array}{lcl}
\mathsf{map}(f, \mathsf{nil}) & = & \mathsf{nil}, \\
\mathsf{map}(f, \mathsf{cons}(x, xs)) & = & \mathsf{cons}(\mathsf{ap}(f, xs), \mathsf{map}(f, xs)) \\
\mathsf{map}(f, \mathtt{BAD}) & = & \mathtt{BAD}, \\
\mathsf{map}(f, x) & = & \mathtt{UNR} \\
\quad \vee\ x = \mathsf{cons}(\mathsf{cons}_0(x), \mathsf{cons}_1(x)) & & \\
\quad \vee\ x = \mathsf{nil} & & \\
\quad \vee\ x = \mathtt{BAD} & &
\end{array}
$$

$$\mathsf{ap}(\mathsf{ap}(\mathsf{map}_{\mathsf{ptr}}, f), xs) = \mathsf{map}(f, xs)$$

Guiding principle for translation of contracts

Theorem

Assume that e and $\mathtt{C}$ contain no free term variables. Then the FOL translation of the claim $e \in \mathtt{C}$ holds in the model if and only if the denotation of e is in the semantics of $\mathtt{C}$. Formally:

$$\langle D_\infty, \mathcal{I} \rangle \models \mathcal{C}\{\!\{e \in \mathtt{C}\}\!\} \quad \Leftrightarrow \quad [\![e]\!] \in [\![\mathtt{C}]\!]$$

Satisfying a Contract, Denotationally

$$[\![\mathtt{C}]\!]_\rho \subseteq D_\infty$$

$$\begin{array}{rcl}
[\![x \mid e]\!]_\rho & = & \{d \mid d = \bot \ \vee \ [\![e]\!]_{\rho,x\mapsto d} \in \{\mathsf{True}, \bot\}\} \\
[\![(x{:}\mathtt{C}_1) \to \mathtt{C}_2]\!]_\rho & = & \{d \mid \forall d' \in [\![\mathtt{C}_1]\!]_\rho.\mathsf{app}(d, d') \in [\![\mathtt{C}_2]\!]_{\rho,x\mapsto d'}\} \\
[\![\mathtt{C}_1 \& \mathtt{C}_2]\!]_\rho & = & \{d|d \in [\![\mathtt{C}_1]\!]_\rho \wedge d \in [\![\mathtt{C}_2]\!]_\rho\} \\
[\![\mathtt{CF}]\!]_\rho & = & F^\infty_{\mathsf{cf}}
\end{array}$$

where

$$\begin{array}{rcl}
F^\infty_{\mathsf{cf}} & = & \{\bot\} \\
& \cup & \{\ \mathsf{K}(\overline{d}) \mid K^n \in \Sigma,\ d_i \in F^\infty_{\mathsf{cf}}\} \\
& \cup & \{\ \mathsf{Fun}(d) \mid \forall d' \in F^\infty_{\mathsf{cf}}.\ d(d') \in F^\infty_{\mathsf{cf}}\}
\end{array}$$

Satisfying a Contract, Denotationally

$$[\![\mathtt{C}]\!]_\rho \subseteq D_\infty$$

$$\begin{array}{rcl}
[\![x \mid e]\!]_\rho & = & \{d \mid d = \bot \lor [\![e]\!]_{\rho,x\mapsto d} \in \{\mathsf{True}, \bot\}\} \\
[\![(x{:}\mathtt{C}_1) \to \mathtt{C}_2]\!]_\rho & = & \{d \mid \forall d' \in [\![\mathtt{C}_1]\!]_\rho . \mathsf{app}(d, d') \in [\![\mathtt{C}_2]\!]_{\rho,x\mapsto d'}\} \\
[\![\mathtt{C}_1 \& \mathtt{C}_2]\!]_\rho & = & \{d | d \in [\![\mathtt{C}_1]\!]_\rho \land d \in [\![\mathtt{C}_2]\!]_\rho\} \\
[\![\mathtt{CF}]\!]_\rho & = & F^\infty_{\mathsf{cf}}
\end{array}$$

where

$$\begin{array}{rcl}
F^\infty_{\mathsf{cf}} & = & \{\bot\} \\
& \cup & \{\ \mathsf{K}(\overline{d}) \mid K^n \in \Sigma,\ d_i \in F^\infty_{\mathsf{cf}}\} \\
& \cup & \{\ \mathsf{Fun}(d) \mid \forall d' \in F^\infty_{\mathsf{cf}}.\ d(d') \in F^\infty_{\mathsf{cf}}\}
\end{array}$$

$$\mathsf{CF}(\mathtt{UNR}),\quad \neg\mathsf{CF}(\mathtt{BAD}),\quad \mathsf{CF}(\mathsf{nil}),$$
$$\mathsf{CF}(\mathsf{cons}(x, xs)) \leftrightarrow (\mathsf{CF}(x) \land \mathsf{CF}(xs))$$

Translating Contracts to FOL

$$
\begin{array}{rcl}
\mathcal{C}\{\!\{e \in \{x \mid p\}\}\!\} & = & \mathcal{E}\{\!\{e\}\!\}{=}\mathtt{UNR} \lor \\
 & & \mathcal{E}\{\!\{p\}\!\}[\mathcal{E}\{\!\{e\}\!\}/x]{=}\mathtt{UNR} \lor \\
 & & \mathcal{E}\{\!\{p\}\!\}[\mathcal{E}\{\!\{e\}\!\}/x]{=}\mathtt{True} \\
 & & \\
\mathcal{C}\{\!\{e \in (x{:}C_1) \to C_2\}\!\} & = & \forall x\,.\,\mathcal{C}\{\!\{x \in C_1\}\!\} \to \mathcal{C}\{\!\{e\;x \in C_2\}\!\} \\
 & & \\
\mathcal{C}\{\!\{e \in C_1 \& C_2\}\!\} & = & \mathcal{C}\{\!\{e \in C_1\}\!\} \land \mathcal{C}\{\!\{e \in C_2\}\!\} \\
 & & \\
\mathcal{C}\{\!\{e \in \mathsf{CF}\}\!\} & = & \mathsf{CF}(\mathcal{E}\{\!\{e\}\!\})
\end{array}
$$

```
contract_1 = head ::: Pred (not . null) --> CF
```

Theorem Prover Queries

We ask for the satisfiablitiy of

$$\mathcal{T}_{\text{datatypes}}, \mathcal{T}_{\text{functions}}, \neg\mathcal{C}\{\!\{e \in C\}\!\}$$

If it is unsatisfiable, we know that

$$\mathcal{T}_{\text{datatypes}}, \mathcal{T}_{\text{functions}} \vdash \mathcal{C}\{\!\{e \in C\}\!\}$$

Carefully designed so the soundness theorem is true :)
What if we get satisfiable?

Recursive functions

```
length []     = Zero
length (x:xs) = Succ (length xs)

length_contract = length ::: CF --> CF
```

Has an counterexample `xs = () : xs`,

$$\texttt{length xs} = \texttt{length (() : xs)} = \texttt{S (length xs)} = \texttt{S inf} = \texttt{inf}$$

Can we have $\neg\mathsf{CF}(\texttt{inf})$? Yes, since the only related axiom says:

$$\neg\mathsf{CF}(\texttt{inf}) \leftrightarrow \neg\mathsf{CF}(\texttt{S inf})$$

Fixed Point Induction

$$\frac{P(\bot) \qquad P(x) \to P(f\ x) \qquad P \text{ admissible}}{P(\mathtt{fix}\ f)}$$

```
length• []     = Zero
length• (x:xs) = Succ (length° xs)
```

$$\frac{P(\mathtt{UNR}) \qquad P(f^{\circ}) \to P(f^{\bullet}) \qquad P \text{ admissible}}{P(f)}$$

$\mathcal{T}_{\text{datatypes}}, \mathcal{T}_{\text{functions}}, \mathcal{C}\{\!\{\mathtt{length}^{\circ} \in \mathsf{CF} \to \mathsf{CF}\}\!\}, \mathcal{C}\{\!\{\mathtt{length}^{\bullet} \notin \mathsf{CF} \to \mathsf{CF}\}\!\}$

Contracts are designed to be admissible predicates.

Infinite models

For a given theory $\mathcal{T}$, either of these three is true:

1. It is unsatisfiable
2. It is *finitely* satisfiable
3. It is only *infinitely* satisfiable

Right now, our axiomatisation typically enforces only infinite models since it has injective and non-surjective functions:

$$\mathsf{just}(x) \neq \mathsf{nothing}, \quad \mathsf{just}_0(\mathsf{just}(x)) = x$$

Quest: find a translation that is either 1 or 2.

Desired properties of an alternative translation

1. *Soundness*

$$\mathcal{T} \vdash \neg(e \notin C) \implies \mathcal{T}^m \vdash \neg(e \notin C)^m$$

2. *Completeness*

$$\mathcal{T}^m \vdash \neg(e \notin C)^m \implies \mathcal{T} \vdash \neg(e \notin C)$$

3. *Finite model guarantees*
   If there exists an M such that:

$$M \models \mathcal{T}, (e \notin C)$$

   then there exists a *finite* M^m such that:

$$M^m \models \mathcal{T}^m, (e \notin C)^m$$

4. *Efficiency*
   The alternative translation is as least as efficient as the original in practice on unsatisfiable theories

“Minimisation”: Our Trick for Finite Models and Efficiency

- Idea: introduce a new predicate, min, that means a term should be subject to reduction (to weak head normal form).
- Selector axioms:

$$\mathsf{min}(\mathsf{just}(x)) \to \mathsf{just}_0(\mathsf{just}(x)) = x$$

- The name comes from that we should try to *minimise* the number of domain elements that are “min”.

Function Translation with Minimisation

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

$$
\begin{array}{lcll}
\mathsf{min}(\mathsf{map}(f,x)) & \rightarrow & \mathsf{map}(f,x) & \\
\mathsf{min}(\mathsf{map}(f,\mathsf{nil})) & \rightarrow & \mathsf{map}(f,\mathsf{nil}) & = \quad \mathsf{nil}, \\
\mathsf{min}(\mathsf{map}(f,\mathsf{cons}(x,xs))) & \rightarrow & \mathsf{map}(f,\mathsf{cons}(x,xs)) & = \\
 & & \quad \mathsf{cons}(\mathsf{ap}(f,xs),\mathsf{map}(f,xs)) & \\
\mathsf{min}(\mathsf{map}(f,\mathtt{BAD})) & \rightarrow & \mathsf{map}(f,\mathtt{BAD}) & = \quad \mathtt{BAD}, \\
\mathsf{min}(\mathsf{map}(f,x)) & \rightarrow & \mathsf{map}(f,x) & = \quad \mathtt{UNR} \\
 & & \quad \vee\ x = \mathsf{cons}(\mathsf{cons}_0(x),\mathsf{cons}_1(x)) & \\
 & & \quad \vee\ x = \mathsf{nil} & \\
 & & \quad \vee\ x = \mathtt{BAD} &
\end{array}
$$

Contract Translation with Minimisation

Distinguish between assumptions ($e \in C$) and goals ($e \notin C$).

Contracts should only be assumed when they are "min", contracts the prove should always be "min" to drive computation.

When doing induction on f, assume for f^{circ} and prove for $f^{\bullet}$.

Ask for the satisfiability of:

$$\mathcal{T}_{\text{datatypes}}, \mathcal{T}_{\text{functions}}, \mathcal{C}\{\!\{f^{\circ} \in C\}\!\}, \mathcal{C}\{\!\{f^{\bullet} \notin C\}\!\}$$

Contract Translation with Minimisation II

$$
\begin{array}{rcl}
\mathcal{C}\{\!\{e \in \{x \mid p\}\}\!\} & = & \mathsf{min}(\mathcal{E}\{\!\{e\}\!\}) \land \mathsf{min}(\mathcal{E}\{\!\{p\}\!\}[\mathcal{E}\{\!\{e\}\!\}/x]) \\
 & & (\mathcal{E}\{\!\{e\}\!\}{=}\mathtt{UNR} \lor \\
 & & \mathcal{E}\{\!\{p\}\!\}[\mathcal{E}\{\!\{e\}\!\}/x]{=}\mathtt{UNR} \lor \\
 & & \mathcal{E}\{\!\{p\}\!\}[\mathcal{E}\{\!\{e\}\!\}/x]{=}\mathtt{True})
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{C}\{\!\{e \notin \{x \mid p\}\}\!\} & = & \mathsf{min}(\mathcal{E}\{\!\{e\}\!\}) \land \mathsf{min}(\mathcal{E}\{\!\{p\}\!\}[\mathcal{E}\{\!\{e\}\!\}/x]) \\
 & & (\mathcal{E}\{\!\{e\}\!\} \neq \mathtt{UNR} \lor \\
 & & \mathcal{E}\{\!\{p\}\!\}[\mathcal{E}\{\!\{e\}\!\}/x]{=}\mathtt{BAD} \lor \\
 & & \mathcal{E}\{\!\{p\}\!\}[\mathcal{E}\{\!\{e\}\!\}/x]{=}\mathtt{False})
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{C}\{\!\{e \in (x{:}C_1) \to C_2\}\!\} & = & \forall x\,.\,\mathsf{min}(e\ x) \to \\
 & & \qquad (\mathcal{C}\{\!\{x \notin C_1\}\!\} \lor \mathcal{C}\{\!\{e\ x \in C_2\}\!\}) \\
\mathcal{C}\{\!\{e \notin (x{:}C_1) \to C_2\}\!\} & = & \exists x\,.\,\mathcal{C}\{\!\{x \in C_1\}\!\} \land \mathcal{C}\{\!\{e\ x \notin C_2\}\!\}
\end{array}
$$

Experimental results

With minimisation:

smt-z3	timeouts:	4.6%	avg:	0.7ms
z3	timeouts:	5.2%	avg:	0.7ms
vampire	timeouts:	19.5%	avg:	17.2ms
equinox	timeouts:	13.8%	avg:	104.1ms
eprover	timeouts:	25.9%	avg:	3.8ms

Without minimisation:

smt-z3	timeouts:	10.3%	avg:	1.8ms
z3	timeouts:	11.5%	avg:	0.5ms
vampire	timeouts:	26.4%	avg:	9.1ms
equinox	timeouts:	45.4%	avg:	23.2ms
eprover	timeouts:	41.4%	avg:	2.4ms

Finite Model Finding

- We use the finite model finder `paradox`, which exhaustively seaches for models with increasing domain size and gives us the smallest possible model.
- Countermodels are typically very few elements (4-6), with many infinite values such as `xs = Nothing : xs`.
- Since constructors now are not injective, we need to do a little work to find out how domain elements really are represented.

Unearthing a Model

```
(-) :: Nat -> Nat -> Nat
x      - Zero   = x
Zero   - _      = error "Negative Nat!"
Succ x - Succ y = x - y
```

$$(\texttt{-}) \in \{\texttt{CF}-> \texttt{CF}-> \texttt{CF}\}$$

paradox gives a countermodel with 5 elements: $\mathbf{D} = \{\mathbf{1}, \mathbf{2}, \cdots, \mathbf{5}\}$

Unearthing a Model

```
(-) :: Nat -> Nat -> Nat
x      - Zero   = x
Zero   - _      = error "Negative Nat!"
Succ x - Succ y = x - y
```

$$(\texttt{-}) \in \{\texttt{CF}- > \texttt{CF}- > \texttt{CF}\}$$

paradox gives a countermodel with 5 elements: $\mathbf{D} = \{\mathbf{1}, \mathbf{2}, \cdots, \mathbf{5}\}$

$$\begin{aligned} \texttt{x} &= \mathbf{3} \\ \texttt{y} &= \mathbf{4} \end{aligned}$$

Figuring out what x and y are

$$
\begin{array}{rclcrclcrcl}
\texttt{x} & = & \mathbf{3} & & \texttt{Succ}(\mathbf{1}) & = & \mathbf{5} & & \texttt{Succ}_0(\mathbf{1}) & = & \mathbf{3} \\
\texttt{y} & = & \mathbf{4} & & \texttt{Succ}(\mathbf{2}) & = & \mathbf{2} & & \texttt{Succ}_0(\mathbf{2}) & = & \mathbf{3} \\
\texttt{BAD} & = & \mathbf{1} & & \texttt{Succ}(\mathbf{3}) & = & \mathbf{4} & & \texttt{Succ}_0(\mathbf{3}) & = & \mathbf{2} \\
\texttt{UNR} & = & \mathbf{2} & & \texttt{Succ}(\mathbf{4}) & = & \mathbf{5} & & \texttt{Succ}_0(\mathbf{4}) & = & \mathbf{3} \\
\texttt{Zero} & = & \mathbf{3} & & \texttt{Succ}(\mathbf{5}) & = & \mathbf{5} & & \texttt{Succ}_0(\mathbf{5}) & = & \mathbf{5}
\end{array}
$$

x	$\texttt{Succ}(x)$	$\texttt{Succ}_0(\texttt{Succ}(x))$
$\mathbf{1}$	$\mathbf{5}$	$\mathbf{5}$
$\mathbf{2}$	$\mathbf{2}$	$\mathbf{3}$
$\mathbf{3}$	$\mathbf{4}$	$\mathbf{3}$
$\mathbf{4}$	$\mathbf{5}$	$\mathbf{5}$
$\mathbf{5}$	$\mathbf{5}$	$\mathbf{5}$

$$\texttt{y} = \texttt{Succ Zero}, \quad \texttt{x} = \texttt{Zero}$$

Ill-typed Models

In the model above, we have

$$\texttt{x} = \texttt{Zero} = \texttt{True}$$

The reason is that we do not add discrimination axioms for elements of different types - these are never needed in proofs.
Two ways to proceed:

- Do type inference on the model to make sure that it is printed type-correct
- Add discrimination axioms for constructors of different types.

Optimisations and tricks

- Inlining: reduces the number of function symbols
- Splitting goals: when proving a contract for a function that is a case expression, generate a theory for each right hand side of the case alternatives
- No native support for integer arithmetic in FOL: use the theory in SMTLIB and use z3

What when we get satisfiable back?

We ask for the satisfiablitiy of

$$\mathcal{T}_{\text{datatypes}}, \mathcal{T}_{\text{functions}}, \neg\phi_{\text{contract}}$$

If it is satisfiable, we know that there exists a model M such that

$$M \models \mathcal{T}_{\text{datatypes}}, \mathcal{T}_{\text{functions}}, \neg\phi_{\text{contract}}$$

Happens when:

- the contract does not hold
- assumptions are missing (induction, other contracts)
- the theory is incomplete

Open Questions / Future Work

- What can we do when a theorem prover says SAT?
- Is there a (provably) complete min-axiomatisation with guaranteed finite countermodels?
- Do we need a theorem prover for (lazy) functional languages?
- `z3`: can triggers be used instead of the `min` predicate?
- `z3`: how to make it prove satisfiable?

Obtaining the contract checker

`github.com/danr/contracts`

unused slides

Contracts

```
head :: [a] -> a
head (x:xs) = x
head []     = error "head: empty list!"
```

Some example contracts for `head`:

$$\texttt{head} \in \mathsf{CF} \rightarrow \mathsf{CF}$$
$$\texttt{head} \in \{\texttt{xs} \mid \texttt{not (null xs)}\} \rightarrow \mathsf{CF}$$
$$\texttt{head} \in \mathsf{CF}\&\{\texttt{xs} \mid \texttt{not (null xs)}\} \rightarrow \mathsf{CF}$$

CF stands for Crash-Free

Splitting Goals

`risers` in GHC Core is a bunch of cases...

```
risers = \ xs -> case xs of {
    [] -> []
    y : ys -> case ys of {
        [] -> [[y]]
        z : zs -> case risers (z:zs) of {
            [] -> error "internal error";
            : s ss -> case y <= z of {
                False -> [y] : (s:ss)
                True ->  (y:s) : ss
    } } } }
```

These cases becomes a big chunk of translated formulae, making a big theory. However, we can split every left-hand side of a case alternative a small, separate theory when proving a contract for `risers`. In practice, these smaller theories are much easier for theorem provers to handle.