Haskell Contract Checking via First-Order Logic

Nathan Collins

Department of Computer Science
Portland State University

RPE Presentation, 11 May 2012

Joint work with Charles-Pierre Astolfi, Koen Claessen, Simon Peyton-Jones, and Dimitrios Vytiniotis
The Haskell type system is powerful:

```
head :: forall t. List t -> t
head xs = case xs of
  Nil    -> error "Empty list!"
  Cons x _ -> x
```

```
head 42 -- Rejected.
```

But it doesn't prohibit exceptions:

```
head Nil :: forall t. t -- Accepted. Uh oh!
```

Contracts to the rescue! Contracts are fancy types:

```
head ::: CF&&{xs | not (null xs)} -> CF
```

Great! But how to check these fancy types? First-order logic to the rescue ... sort of.
The Haskell type system is powerful:

```haskell
head :: forall t. List t -> t
head xs = case xs of
  Nil -> error "Empty list!"
  Cons x _ -> x
```

head 42 -- Rejected.

But it doesn’t prohibit exceptions:

```
head Nil :: forall t. t -- Accepted. Uh oh!
```

Contracts to the rescue! Contracts are fancy types:

```
head :: CF && \{xs | not (null xs)\} -> CF
```

Great! But how to check these fancy types? First-order logic to the rescue ... sort of.
Introduction

The Haskell type system is powerful:

```haskell
head :: forall t. List t -> t
head xs = case xs of
  Nil     -> error "Empty list!"
  Cons x _ -> x
```

```
head 42 -- Rejected.
```

But it doesn’t prohibit exceptions:

```
head Nil :: forall t. t -- Accepted. Uh oh!
```

Contracts to the rescue! Contracts are fancy types:

```haskell
head :::: CF&&{xs | not (null xs)} -> CF
```

Great! But how to check these fancy types? First-order logic to the rescue ... sort of.
The Haskell type system is powerful:

```haskell
head :: forall t. List t -> t
head xs = case xs of
    Nil     -> error "Empty list!"
    Cons x _ -> x
```

```
head 42 -- Rejected.
```

But it doesn’t prohibit exceptions:

```
head Nil :: forall t. t -- Accepted. Uh oh!
```

Contracts to the rescue! Contracts are fancy types:

```
head ::: CF&&{xs | not (null xs)} -> CF
```

Great! But how to check these fancy types?

First-order logic to the rescue ... sort of.
The Haskell type system is powerful:

```haskell
head :: forall t. List t -> t
head xs = case xs of
  Nil    -> error "Empty list!"
  Cons x _ -> x
```

```haskell
head 42 -- Rejected.
```

But it doesn’t prohibit exceptions:

```haskell
head Nil :: forall t. t -- Accepted. Uh oh!
```

Contracts to the rescue! Contracts are fancy types:

```haskell
head :::: CF&&{xs | not (null xs)} -> CF
```

Great! But how to check these fancy types? First-order logic to the rescue ... sort of.
Goal: *effective* static contract checking.

Overview of Contracts

Checking Contracts: Translating Haskell to FOL

Experiments

Conclusions/Future Work
My Contributions

- Rewrote the contract checker and added many features.
- Designed and implemented the Min-translation.
- Wrote many examples, including the first use of lemmas.
- Designed and implemented a type checker for contracts.
- ...and now: documented the research in an RPE paper.
Data:

\[
[0, 1, 2] \\
= \text{Cons } 0 \text{ (Cons } 1 \text{ (Cons } 2 \text{ Nil))} \\
= \text{Cons } Z \text{ (Cons } (S \text{ Z)} \text{ (Cons } (S \text{ (S Z)) Nil))}
\]

Judgments:

- Has type: \( e :: t \)
- Has contract: \( e :::: c \)
An Example Contract

\[ c ::= \text{CF} \quad -- \text{Crash free} \\
| \text{c} \&\& \text{c} \quad -- \text{Conjunction} \\
| \text{c} \mid \mid \text{c} \quad -- \text{Disjunction} \\
| \text{x:} \text{c} \rightarrow \text{c} \quad -- \text{Implication} \\
| \{\text{x|p}\} \quad -- \text{Refinement} \]

Example: CF is not a syntactic property:

\[
\begin{align*}
\text{fst (x,}_-\text{)} &= x \\
\text{snd (_,}y\text{)} &= y
\end{align*}
\]

1. \(\text{fst (Z, error "Oh no!") :::: CF}\).
2. But not \(\text{(Z, error "Oh no!") :::: CF}\), because \(\text{snd (Z, error "Oh no!")}\) is a crash.
An Example Contract

\[
c ::= \text{CF} \quad \text{-- Crash free}
| \ c \&\& \ c \quad \text{-- Conjunction}
| \ c \mid\mid \ c \quad \text{-- Disjunction}
| \ x: c \rightarrow c \quad \text{-- Implication}
| \ \{x \mid p\} \quad \text{-- Refinement}
\]

Example: CF is not a syntactic property:

\[
\begin{align*}
\text{fst} (x, \_ & \phantom{= x}) = x \\
\text{snd} (\_, y & \phantom{= x}) = y
\end{align*}
\]

1. \[\text{fst} (\text{Z, error "Oh no!") ::: CF.}\]
2. But not \(\text{(Z, error "Oh no!") ::: CF}\), because \[\text{snd} (\text{Z, error "Oh no!") is a crash.}\]
Another Example Contract

c ::= CF -- Crash free
| c&&c -- Conjunction
| c||c -- Disjunction
| x:c -> c -- Implication
| {x|p} -- Refinement

Example: refinement, implication, and conjunction:

lookUp :: forall t. Nat -> List t -> t
lookUp n xs = case xs of
    Nil -> error "List is too short!"
    Cons x xs’ -> case n of
        Z -> x
        S n’ -> lookUp n’ xs’
lookUp ::: n:CF -> ({xs|n < length xs}&&CF) -> CF
Contracts Are Useful

- Static type checking = compile-time approximation to run-time program behavior.
- Contracts + types = better approximation.

\[
\text{sort} :: \forall t. \text{List } t \rightarrow \text{List } t
\]

\[
\text{sort} :::: \text{CF} \rightarrow \text{CF} \&\& \{xs| \text{sorted } xs\}
\]
Contracts Are Useful . . . But Difficult to Check Statically

```
error :: forall t. String -> t
head xs = case xs of
  Nil    -> error "Empty list!"
  Cons x _ -> x
```

Type checking is path *insensitive* (easy):

```
head :: forall t. List t -> t
```

Contract checking is path *sensitive*:

```
head :: CF&&{xs | not (null xs)} -> CF
```

And must reason about arbitrary computations (undecidable):

```
not (null xs) = True  ⇒  xs ≠ Nil
```
error :: forall t. String -> t
head xs = case xs of
  Nil     -> error "Empty list!"
  Cons x _ -> x

Type checking is path \textit{insensitive} (easy):

\[
\text{head} :: \forall t. \text{List } t \to t
\]

Contract checking is path \textit{sensitive}:

\[
\text{head} :: \text{CF} \&\& \{ \text{xs | not (null xs)} \} \to \text{CF}
\]

And must reason about arbitrary computations (undecidable):

\[
\text{not (null xs)} = \text{True} \implies \text{xs \neq Nil}
\]
Naive translation of \( \text{map} \)'s definition:

\[
\forall f \, xs. \quad (xs = \text{Nil}) \rightarrow (\text{map} \, f \, xs = \text{Nil}) \\
\wedge \quad \forall x \, xs'. \\
\quad (xs = \text{Cons} \, x \, xs') \rightarrow \\
\quad (\text{map} \, f \, xs = \text{Cons} \, (f \, x) \, (\text{map} \, f \, xs')) \\
\vdots \\
\wedge \quad (xs = \text{Nil}) \lor (\exists x \, xs'. \, xs = \text{Cons} \, x \, xs') \lor \cdots
\]
The Naive Translation . . . is Naive

- **Problem**: prover wastes time on pointless instantiations.

Naive translation of map’s definition (unchanged):

\[
\forall f \; xs. \; (xs = \text{Nil}) \rightarrow (\text{map } f \; xs = \text{Nil}) \\
\land \; \forall \; x \; xs'. \\
\quad (xs = \text{Cons } x \; xs') \rightarrow \quad (\text{map } f \; xs = \text{Cons } (f \; x) \; (\text{map } f \; xs')) \\
\vdots \\
\land \; (xs = \text{Nil}) \lor (\exists x \; xs'. \; xs = \text{Cons } x \; xs') \lor \cdots
\]
The Less-Naive Translation

- **Problem:** prover wastes time on pointless instantiations.
- **Solution:**
  - **Idea:** restrict instantiation to “interesting” terms.
  - **Implementation:** “\( \text{Min}(e) \)” means “\( e \) is interesting”.

Less-naive translation of `map`’s definition:

\[
\forall f \, xs. \quad \text{Min}(
\text{map} \, f \, xs) \rightarrow 
\begin{array}{l}
(xs = \text{Nil}) \rightarrow (\text{map} \, f \, xs = \text{ Nil}) \\
\wedge \quad \forall x \, xs'. \\
\qquad (xs = \text{Cons} \, x \, xs') \rightarrow \\
\qquad (\text{map} \, f \, xs = \text{Cons} \, (f \, x) \, (\text{map} \, f \, xs')) \\
\vdots \\
\wedge (xs = \text{Nil}) \vee (\exists x \, xs'. \, xs = \text{Cons} \, x \, xs') \vee \cdots \\
\wedge \quad \text{Min}(xs) 
\end{array}
\]
How to Design a Less-Naive Translation

- Restrict prover’s search space using $\text{Min}$.  
- Evaluation semantics + axiom/goal distinction motivate $\text{Min}$ placement.

See paper for details.
Experiments: Running-time Comparison

![Histogram of running times for various experiments.](image-url)
Conclusion

Progress made:

- Adding `Min` significantly improves performance.

But lots of room for improvement:

- Debugging failed proofs is hard:
  - Is the contract wrong?
  - Or are the axioms insufficient?
- Need better feedback from contract checker:
  - Which part of which contract is violated?
  - What execution path leads to violation?
- Need better lemma support:
  - Lemma use shouldn’t affect run-time behavior.
  - Equational reasoning would help.
Future Work

Improve contract checker:

- Better feedback on failure by making goals:
  - Smaller: \((\phi \rightarrow \bigwedge_i \phi_i) \equiv \bigwedge_i (\phi \rightarrow \phi_i)\)
  - Path-based.

- More expressive proof system:
  - Real lemmas?
  - Structural (co-)induction?

- More expressive contract system:
  - Equality?
  - **Contract polymorphism.**
  - Constructor contracts.
  - **Recursive contract definitions.**

```haskell
data List t = Nil | Cons t (List t)
contract ListC c = Nil || Cons c (ListC c)
map :: forall s t. (s -> t) -> List s -> List t
map ::: forall c d. (c -> d) -> ListC c -> ListC d
```