Haskell Contract Checking via First-Order Logic

Nathan Collins [1]

Department of Computer Science
Portland State University

RPE Presentation, 11 May 2012

[1] Joint work with Charles-Pierre Astolfi, Koen Claessen, Simon Peyton-Jones, and Dimitrios Vytiniotis

Introduction

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.

Introduction

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.

Introduction

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.

Introduction

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.

Introduction

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.

Outline

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.

Notation

Data:

```
[0,1,2]
  = Cons 0 (Cons   1   (Cons     2     Nil))
  = Cons Z (Cons (S Z) (Cons (S (S Z)) Nil))
```

Judgments:

- Has type: `e :: t`
- Has contract: `e ::: c`

An Example Contract

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

Example: CF is not a syntactic property:

```
fst (x,_) = x
snd (_,y) = y
```

1. `fst (Z, error "Oh no!") ::: CF`.
2. But not `(Z, error "Oh no!") ::: CF`, because `snd (Z, error "Oh no!")` is a crash.

An Example Contract

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

Example: CF is not a syntactic property:

```
fst (x,_) = x
snd (_,y) = y
```

1. `fst (Z, error "Oh no!") ::: CF`.
2. But not `(Z, error "Oh no!") ::: CF`, because `snd (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.

```
sort ::  forall t. List t -> List t
sort ::: CF -> CF&&{xs|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):

$$\texttt{not (null xs)} = \texttt{True} \quad \Longrightarrow \quad \texttt{xs} \neq \texttt{Nil}$$

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
```

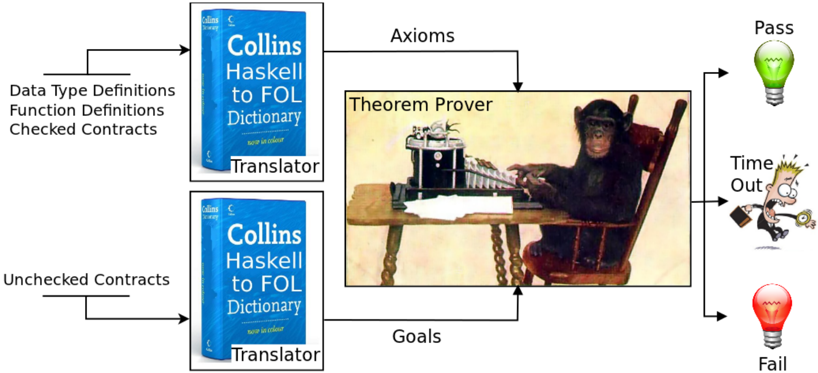
Contract checking is path *sensitive*:

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

And must reason about arbitrary computations (undecidable):

$$\texttt{not (null xs)} = \texttt{True} \quad \Longrightarrow \quad \texttt{xs} \neq \texttt{Nil}$$

Contract Checking Process



The Naive Translation

```
map ::: (CF -> CF) -> CF -> CF
map :: forall s t. (s -> t) -> List s -> List t
map f xs = case xs of
  Nil          -> Nil
  Cons x xs' -> Cons (f x) (map f xs')
```

Naive translation of map's definition:

$$
\begin{aligned}
\forall\, \texttt{f xs}. \quad & (\texttt{xs} = \texttt{Nil}) \rightarrow (\texttt{map f xs} = \texttt{Nil}) \\
\wedge \quad & \forall\, \texttt{x xs'}. \\
& \quad (\texttt{xs} = \texttt{Cons x xs'}) \rightarrow \\
& \quad (\texttt{map f xs} = \texttt{Cons (f x) (map f xs')}) \\
\vdots \quad & \\
\wedge \quad & (\texttt{xs} = \texttt{Nil}) \vee (\exists\, \texttt{x xs'}.\ \texttt{xs} = \texttt{Cons x xs'}) \vee \cdots
\end{aligned}
$$

The Naive Translation . . . is Naive

- Problem: prover wastes time on pointless instantiations.

Naive translation of map's definition (unchanged):

$$
\begin{aligned}
\forall\ \mathtt{f\ xs}.\quad & (\mathtt{xs} = \mathtt{Nil}) \rightarrow (\mathtt{map\ f\ xs} = \mathtt{Nil}) \\
\wedge\quad & \forall\ \mathtt{x\ xs'}. \\
& \quad (\mathtt{xs} = \mathtt{Cons\ x\ xs'}) \rightarrow \\
& \quad (\mathtt{map\ f\ xs} = \mathtt{Cons\ (f\ x)\ (map\ f\ xs')}) \\
\vdots \\
\wedge\quad & (\mathtt{xs} = \mathtt{Nil}) \vee (\exists\ \mathtt{x\ xs'}.\ \mathtt{xs} = \mathtt{Cons\ x\ xs'}) \vee \cdots
\end{aligned}
$$

The Less-Naive Translation

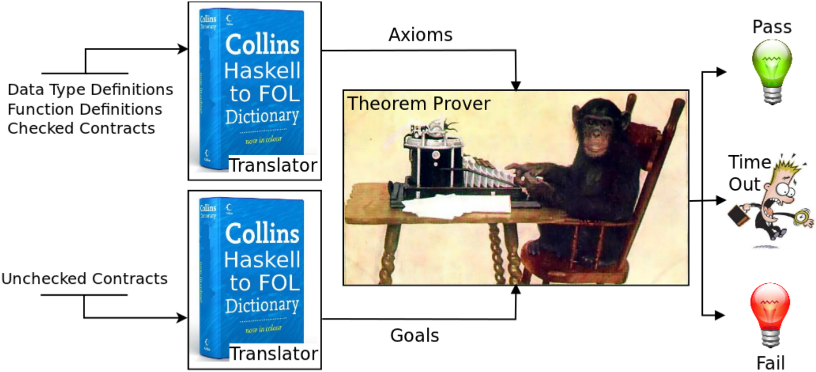
- Problem: prover wastes time on pointless instantiations.
- Solution:
  - Idea: restrict instantiation to "interesting" terms.
  - Implementation: "`Min(e)`" means "e is interesting".

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

$$
\begin{aligned}
\forall\ \mathtt{f}\ \mathtt{xs}.\quad & \mathtt{Min(map\ f\ xs)} \rightarrow \Big(\\
& (\mathtt{xs} = \mathtt{Nil}) \rightarrow (\mathtt{map\ f\ xs} = \mathtt{Nil}) \\
\wedge\quad & \forall\ \mathtt{x}\ \mathtt{xs'}. \\
& \quad (\mathtt{xs} = \mathtt{Cons\ x\ xs'}) \rightarrow \\
& \quad (\mathtt{map\ f\ xs} = \mathtt{Cons\ (f\ x)\ (map\ f\ xs')}) \\
\vdots\quad & \\
\wedge\quad & (\mathtt{xs} = \mathtt{Nil}) \vee (\exists\ \mathtt{x}\ \mathtt{xs'}.\ \mathtt{xs} = \mathtt{Cons\ x\ xs'}) \vee \cdots \\
\wedge\quad & \mathtt{Min(xs)} \Big)
\end{aligned}
$$

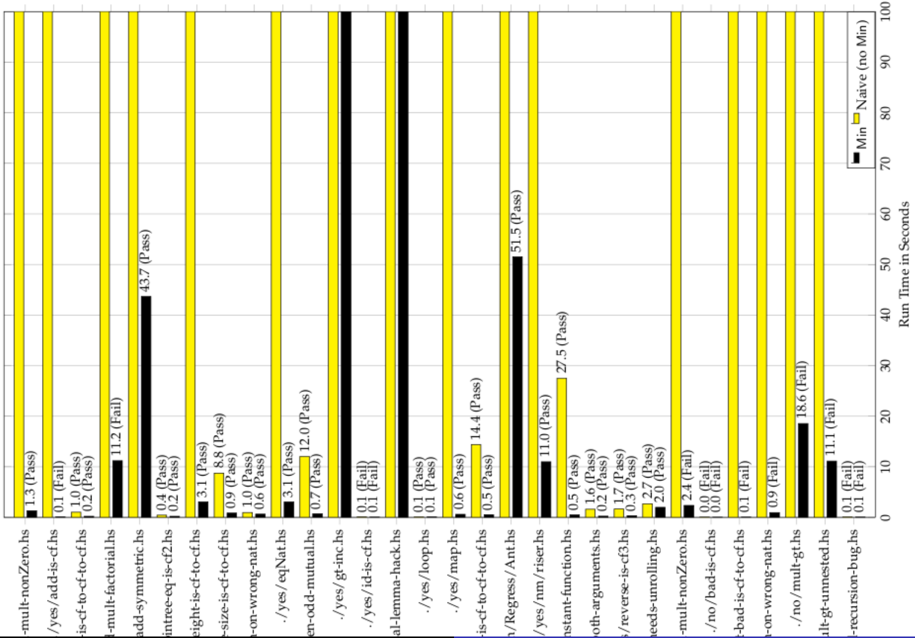
How to Design a Less-Naive Translation

- Restrict prover's search space using `Min`.
- Evaluation semantics + axiom/goal distinction motivate `Min` placement.



See paper for details.

Experiments: Running-time Comparison



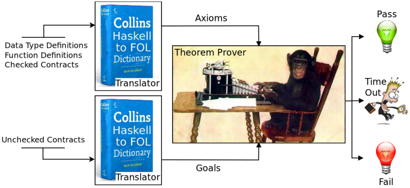
Conclusion

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: $(\phi \to \bigwedge_i \phi_i) \equiv \bigwedge_i (\phi \to \phi_i)$
  - Path-based.
- More expressive proof system:
  - Real lemmas?
  - Structural (co-)induction?
- More expressive contract system:
  - Equality?
  - Contract polymorphism.
  - Constructor contracts.
  - Recursive contract definitions.

```
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
```