Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Inferring Contracts for Functional Programs

Thesis defense

Jurriën Stutterheim

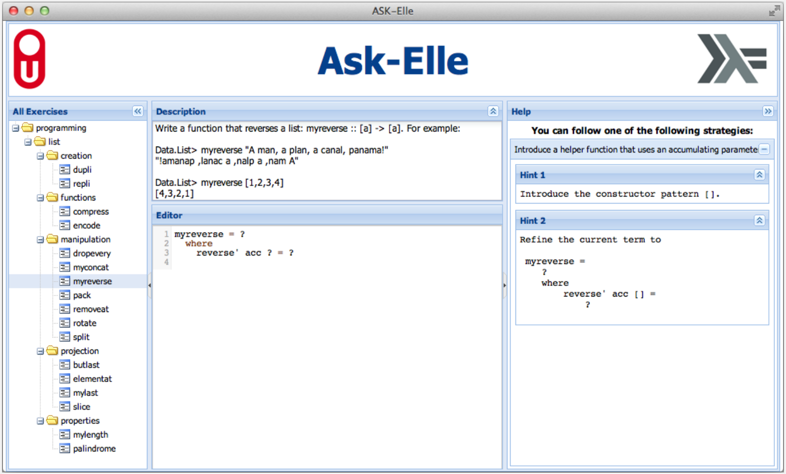
24 January 2013

Ask-Elle: A programming tutor for Haskell

Gerdes et al. are developing a programming tutor for Haskell. Using the tutor, a student can:

- develop her program incrementally
- receive feedback about whether or not she is on the right track
- can ask for a hint when she is stuck
- see how a complete program is constructed step by step

The tutor targets first-year computer science students.


ASK-Elle
Ask-Elle
All Exercises
programming
list
creation
dupli
repli
functions
compress
encode
manipulation
dropevery
myconcat
myreverse
pack
removeat
rotate
split
projection
butlast
elementat
mylast
slice
properties
mylength
palindrome
Description
Write a function that reverses a list: myreverse :: [a] -> [a]. For example:
Data.List> myreverse "A man, a plan, a canal, panama!"
"!amanap ,lanac a ,nalp a ,nam A"
Data.List> myreverse [1,2,3,4]
[4,3,2,1]
Editor
myreverse = ?
where
reverse' acc ? = ?
Help
You can follow one of the following strategies:
Introduce a helper function that uses an accumulating paramete
Hint 1
Introduce the constructor pattern [].
Hint 2
Refine the current term to
myreverse =
?
where
reverse' acc [] =
?

Main ideas behind Ask-Elle

- A teacher specifies model solution solutions for an exercise
- Ask-Elle compares possibly partial student solutions against model solutions using strategies
- As long as a student follows a model solution, Ask-Elle can give hints

And what if a student makes an error?

```
You have made a, possibly incorrect,
step that does not follow the strategy.
```

Our goal

- Specify the properties a solution should satisfy
- Test the properties using QuickCheck
- Express the properties a solution should satisfy as a contract
- Use contract inference to infer contracts for user-defined functions
- Use contract checking to report property violations as precisely as possible

This talk

- Gives a quick introduction to QuickCheck and contracts
- Presents contract inference
- Touches on the current limitations of contract inference

QuickCheck

- A library for property-based testing of programs
- Programmer specifies properties a function has to adhere to
- QuickCheck generates random values and applies the property to them
- If the property fails, QuickCheck tries to shrink the random value to produce a minimal counter-example

QuickCheck: an example

$$propEven : Int \to Bool$$
$$propEven\ n = even\ n$$

```
*Main> quickCheck propEven
*** Failed! Falsifiable (after ...):
1
```

QuickCheck and Ask-Elle

- Specify QuickCheck properties for exercises
- When strategies fail, fall back to QuickCheck
- Problem: *where* is the error?

Contracts

- Impose restrictions and give guarantees for programs
- If a contract is violated, an exception is thrown containing the location of the violation
  - Easier debugging
  - Easier runtime enforcement of invariants
- We use the `typed-contracts` library by Hinze et al.

Specifying contracts

Contracts are represented by a GADT:

$$
\begin{array}{l}
\textbf{data } \mathit{Contract}\ a \textbf{ where} \\
\quad \mathit{Prop} \qquad : \ (a \to \mathit{Bool}) \to \mathit{Contract}\ a
\end{array}
$$

Specifying contracts

Contracts are represented by a GADT:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Contract}\ a\ \mathbf{where}\\
&\quad \mathit{Prop} \quad\ : (a \to \mathit{Bool}) \to \mathit{Contract}\ a\\
&\quad \mathit{Function} : \mathit{Contract}\ a \to \mathit{Contract}\ b\\
&\qquad\qquad\quad \to \mathit{Contract}\ (a \to b)
\end{aligned}
$$

Specifying contracts

Contracts are represented by a GADT:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \mathit{Contract}\ a\ \textbf{where}} \\
\quad \mathit{Prop} & : & (a \to \mathit{Bool}) \to \mathit{Contract}\ a \\
\quad \mathit{Function} & : & \mathit{Contract}\ a \to \mathit{Contract}\ b \\
 & \to & \mathit{Contract}\ (a \to b) \\
\quad \mathit{And} & : & \mathit{Contract}\ a \to \mathit{Contract}\ a \to \mathit{Contract}\ a
\end{array}
$$

Specifying contracts

Contracts are represented by a GADT:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data } \mathit{Contract}\ a \textbf{ where}} \\
\quad \mathit{Prop} & : & (a \to \mathit{Bool}) \to \mathit{Contract}\ a \\
\quad \mathit{Function} & : & \mathit{Contract}\ a \to \mathit{Contract}\ b \\
 & \to & \mathit{Contract}\ (a \to b) \\
\quad \mathit{And} & : & \mathit{Contract}\ a \to \mathit{Contract}\ a \to \mathit{Contract}\ a \\
\quad \mathit{List} & : & \mathit{Contract}\ a \to \mathit{Contract}\ [a] \\
\quad \mathit{Pair} & : & \mathit{Contract}\ a \to \mathit{Contract}\ b \to \mathit{Contract}\ (a, b)
\end{array}
$$

Specifying contracts

Contracts are represented by a GADT:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \mathit{Contract}\ a\ \textbf{where}} \\
\quad \mathit{Prop} & : & (a \to \mathit{Bool}) \to \mathit{Contract}\ a \\
\quad \mathit{Function} & : & \mathit{Contract}\ a \to \mathit{Contract}\ b \\
 & \to & \mathit{Contract}\ (a \to b) \\
\quad \mathit{And} & : & \mathit{Contract}\ a \to \mathit{Contract}\ a \to \mathit{Contract}\ a \\
\quad \mathit{List} & : & \mathit{Contract}\ a \to \mathit{Contract}\ [a] \\
\quad \mathit{Pair} & : & \mathit{Contract}\ a \to \mathit{Contract}\ b \to \mathit{Contract}\ (a, b) \\
\quad \mathit{Functor} & : & \mathit{Functor}\ f \Rightarrow \mathit{Contract}\ a \to \mathit{Contract}\ (f\ a) \\
\quad \mathit{Bifunctor} & : & \mathit{Bifunctor}\ f \Rightarrow \mathit{Contract}\ a \to \mathit{Contract}\ b \\
 & \to & \mathit{Contract}\ (f\ a\ b)
\end{array}
$$

Asserting contracts

$$assert : Contract\ a \rightarrow a \rightarrow a$$

$assert$ is a partial identity: if the contract is satisfied, it acts as identity, if not, it throws an exception.

Notation

$$
\begin{aligned}
\{p\} &= \mathit{Prop}\ p \\
c_1 \twoheadrightarrow c_2 &= \mathit{Function}\ c_1\ c_2 \\
(\&) &= \mathit{And} \\
[c] &= \mathit{List}\ c \\
c_1 \ \langle @ \rangle \ \ c_2 &= c_1\ \&\ \mathit{Functor}\ c_2 \\
c_1 \ \langle @@ \rangle \ (c_2, c_3) &= c_1\ \&\ \mathit{Bifunctor}\ c_2\ c_3
\end{aligned}
$$

Asserting contracts: example

$$f : Int \to Int$$
$$f = assert\ (nat \rightarrowtriangle nat)\ (+1)$$

```
*Main> f 1
2
*Main> f (-1)
*** Exception: 'f' is to blame
```

An example problem in Ask-Elle

```
Write a function that sorts a list

sort :: Ord a => [a] -> [a]

For example:

Data.List> sort [1,2,1,3,2,4]
[1,1,2,2,3,4]
```

Sorting: a model solution

$$
\begin{aligned}
&\mathit{sort} = \mathit{foldr}\ \mathit{insert}\ [\,] \\
&\mathit{insert}\ x\ [\,] && = [x] \\
&\mathit{insert}\ x\ (y :: ys) \mid x \leqslant y && = x :: y :: ys \\
&\phantom{\mathit{insert}\ x\ (y :: ys)} \mid \mathit{otherwise} && = y :: \mathit{insert}\ x\ ys
\end{aligned}
$$

An error in sorting

$$sort' = foldr\ insert'\ [\,]$$

$$\begin{aligned}
&insert'\ x\ [\,] && = [x] \\
&insert'\ x\ (y :: ys) \mid x \leqslant y && = y :: x :: ys \\
& \mid otherwise && = y :: insert'\ x\ ys
\end{aligned}$$

Sorting: a property

$$propSort\ xs = isNonDesc\ (sort'\ xs)$$

$$\begin{aligned} isNonDesc\ (x :: y :: xs) &= x \leqslant y \wedge isNonDesc\ (y :: xs) \\ isNonDesc\ \_ &= True \end{aligned}$$

For the astute observer: no, this property does not fully cover what it means for a list to be sorted, but please bear with me

Running QuickCheck

```
*Main> quickCheck propSort
*** Failed! Falsifiable (after ...):
[0,1]
```

But where is the error?

Contracts...

Contracts to the rescue.

A contracted sort

$$
\begin{array}{l}
sortc = assert \\
\quad ([true] \multimap \{ isNonDesc \}) \\
\quad (\lambda xs \rightarrow sort'\ xs)
\end{array}
$$

Blaming

```
*Main> sortc [0,1]
*** Exception: contract failed:
the expression ‘sort’ is to blame.
```

But where is the error?

A more precise location

To get a more precise location for the error: replace all functions in the definition of $sort$ by their contracted counterparts.

A contracted *insert*

$$
\begin{array}{l}
insertc = assert \\
\quad (true \twoheadrightarrow \{\, isNonDesc \,\} \twoheadrightarrow \{\, isNonDesc \,\}) \\
\quad (\lambda x \rightarrow \lambda xs \rightarrow insert'\ x\ xs)
\end{array}
$$

Blaming II

```
*Main> sortc [0,1]
*** Exception: contract failed:
the expression ‘insert’ is to blame.
```

But wait

- In the context of the tutor, we only know the contract for *sort*
- A student can implement *sort* in many different ways
- We want to infer the contracts for the components of a function

Problem:

Given a well-typed program, determine the contracts for the components of the function.

Inferring contracts

- We have developed a contract inference algorithm
- Based on Algorithm $\mathcal{W}$ by Damas and Milner
- We call it Algorithm $\mathcal{CW}$
- We have implemented Algorithm $\mathcal{CW}$ for a small, **let**-polymorphic lambda-calculus based language with several built-in data types
  - We expect the results to carry over to Haskell
  - We use Haskell in these slides

Predefined contracts

- $true$: never fails assertion
- $false$: always fails assertion
- $list$: succeeds for lists
- $maybe$: succeeds for $Maybe$ values
- $pair$: succeeds for pairs
- $either$: succeeds for $Either$ values
- int: succeeds for integers
- $bool$: succeeds for booleans
- $char$: succeeds for characters

A contract for id

For

- $id : a \to a$

we infer

- $true \rightharpoondown true$

Instead of type variables, we infer $true$ contracts

Inferred contract for *const*

For

- $const : a \to b \to a$

we infer

- $true_1 \rightharpoonup true_2 \rightharpoonup true_1$

Inferred contract for *map*

For

- $map : (a \rightarrow b) \rightarrow [\,a\,] \rightarrow [\,b\,]$

we infer

$$(true_1 \rightarrowtriangle true_2) \rightarrowtriangle (list_1 \lessdot@\gtrdot\; true_1) \rightarrowtriangle (list_2 \lessdot@\gtrdot\; true_2)$$

Types and contracts

- The same type variable gets the same $true$ contract
- For concrete types we introduce fresh specific contracts
- For (bi)functorial types we infer the more general (bi)functor contracts
- We infer the most specific contracts that never fails assertion

Contract unification

Contracts can be unified, creating substitutions:

$$\mathcal{U}\ (true, char) = [true \mapsto char]$$

$$\begin{aligned}\mathcal{U}\ (true_1 \rightarrowtriangle true_2, int \rightarrowtriangle bool) = [&true_2 \mapsto bool \\ ,&true_1 \mapsto int]\end{aligned}$$

Contract unification

We can also unify two non-$true$ contracts:

$$\mathcal{U}\ (int, nat) = [\,int \mapsto nat\,]$$

This is essential, because we want assertion to be able to fail.

Why can we do this?

Contract semantics as sets

The semantics of a contract c, written $[\![c]\!]$, is defined as the set of Haskell values for which it never fails assertion.

For all contracts c, $[\![false]\!] \subseteq [\![c]\!] \subseteq [\![true]\!]$

Unification

Unification of two different contracts is defined as

$$\mathcal{U}\ (c_1, c_2) = [c_1 \mapsto c_2]\ (\textit{iff}\ c_1 \notin ftc(c_2) \wedge [\![c_2]\!] \subseteq [\![c_1]\!])$$
$$\mathcal{U}\ (c_1, c_2) = [c_2 \mapsto c_1]\ (\textit{iff}\ c_2 \notin ftc(c_1) \wedge [\![c_1]\!] \subseteq [\![c_2]\!])$$

$ftc(c)$ is the set of free $true$ contracts in c and $c_1 \notin ftc(c_2)$ is the occurs check.

Since $int \notin ftc(nat) \wedge [\![nat]\!] \subseteq [\![int]\!]$

$$\mathcal{U}\ (int, nat) = [int \mapsto nat]$$

Dependent contracts: sorting revisited

Sorting not just returns a non-descending list, it is also a permutation of the input:

$$isSorted\ xs\ ys = isNonDesc\ ys \wedge ys \in permutations\ xs$$

The contract for the output depends on the input of the function; it is a dependent contract

Defining dependent contracts

To model dependent contracts, we modify the *Contract* GADT

$$
\begin{aligned}
&\textbf{data}\ \mathit{Contract}\ a\ \textbf{where}\\
&\qquad ..\\
&\quad \mathit{Function}\ :\ \mathit{Contract}\ a \to (a \to \mathit{Contract}\ b)\\
&\qquad\qquad\quad \to \mathit{Contract}\ (a \to b)\\
&\qquad ..
\end{aligned}
$$

and introduce new notation:

$$
\begin{aligned}
c_1 \rightarrowtriangle\ c_2 &= \mathit{Function}\ c_1\ (\mathit{const}\ c_2)\\
c_2 \xmapsto{d} c_2 &= \mathit{Function}\ c_1\ c_2
\end{aligned}
$$

Dependent contract for sorting

$$sortContract = (xs : list \lessdot @ \gtrdot true) \xmapsto{d} \{ isSorted\ xs \}$$

In general, if at all possible, inferring and correctly unifying dependent contracts is hard, so we do not attempt it

Working around dependent contracts

- We can eliminate the need for dependent contracts by inling a QuickCheck counter-example in the contract
- QuickCheck shrinks its counter-example
  - We know smaller values will not violate the contract
  - We know that the counter-example will violate the contract

Eliminating dependent contracts: *sort*

Using counter-example $[0,1]$:

$$
\begin{aligned}
\mathit{sortCntr}' &= (\mathit{list} \lessdot\!@\!\gtrdot\ \mathit{true}) \rightarrow \{ \mathit{isSorted}\ [0,1] \} \\
\mathit{insertCntr}' &= \mathit{true} \rightarrow \mathit{true} \rightarrow \{ \mathit{isSorted}\ [0,1] \} \\
\mathit{sortc}' &= \mathit{assert}\ \mathit{sortCntr}'\ \mathit{sort}' \\
\mathit{insertc}' &= ..
\end{aligned}
$$

```
*Main> sortc' [0,1]
*** Exception: contract failed:
the expression 'insert' is to blame.
```

N.B.: we must only assert $\mathit{insertCntr}'$ once, because the contract will now always fail for smaller lists in recursive calls

Future work: eliminating dependent contracts

- Inlining works for inductive types, because they have a base-case
- By default, QuickCheck doesn't actually guarantee a minimal counter-example, but this is easily implemented
- What about non-inductive (flat) types, like Int, $Integer$ and $Char$?

Future work: constant expression contracts

For

$$f\ x = 1$$

we could infer a constant expression contract

$$true \rightarrow \{== 1\}$$

Can we always infer constant expression contracts?

Future work: constant expression contracts

- Under-explored aspect of contract inference
- In some cases, unifying constant expression contracts leads to wrong contracts for sub-expressions
- Can we modify Algorithm $\mathcal{CW}$ to infer constant expression contracts and unify them correctly?

Future work: overview

- Make the dependent contract workaround work for non-inductive types
  - Or find a way to correctly infer dependent contracts
- Correctly infer and unify constant expression contracts
- Implement contract inference in ASK-ELLE

Conclusions

- We can infer and unify non-dependent contracts easily
- For inductive types we can eliminate the need for dependent contracts by inlining a QuickCheck counter-example
- For non-inductive types, it is unclear whether we can eliminate dependent contracts
- We can infer constant expression contracts easily, but unifying them correctly may be more difficult