Universiteit Utrecht

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

Contract Inference for the Ask-Elle Programming Tutor

Master thesis defense under the supervision of Johan Jeuring

Beerend Lauwers

26 February 2014

Outline

1. The Ask-Elle programming tutor

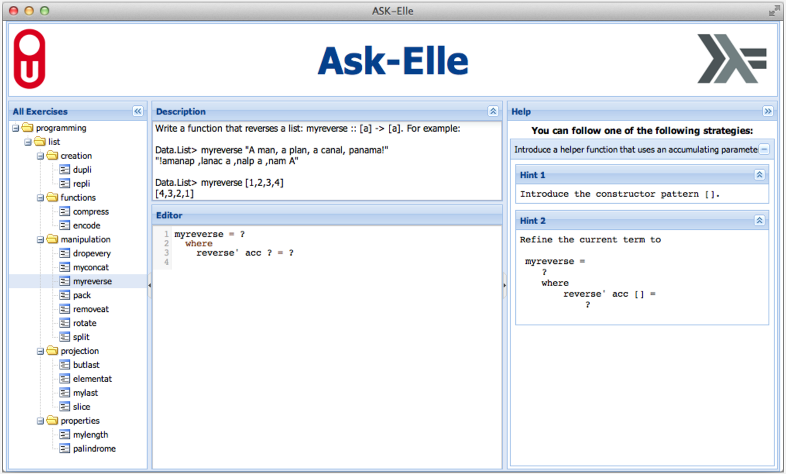
Ask-Elle?

- A web-based programming tutor for Haskell
- Developed by Alex Gerdes for his PhD
- Aims to help first-year CS students

How it works:

- A student selects an exercise and Ask-Elle describes the goal
- Student writes the program incrementally, leaving holes
- Ask-Elle understands the student's progress and can provide feedback
- Student can ask for hints

Screenshot of Ask-Elle interface



Behind the scenes of Ask-Elle

- To define an exercise, a teacher provides model solutions.
- Using strategies, Ask-Elle compares a student's code against these model solutions
- If a student's code can be reduced to a model solution, Ask-Elle can provide detailed feedback and hints
- What happens when the student *doesn't* follow a model solution?

A gap in detailed feedback

No model solution fits the student's solution? QuickCheck!

```
"Wrong solution:
 range 4 6 provides a counterexample."
```

Can we provide richer feedback and offer a more precise location of the programming error?

A gap in detailed feedback

No model solution fits the student's solution? QuickCheck!

```
"Wrong solution:
 range 4 6 provides a counterexample."
```

Can we provide richer feedback and offer a more precise location of the programming error? Yes, with contracts!

2. Contracts

What's a contract?

Just like its real-world counterpart, a programming contract stipulates prerequisites and guarantees between two parties:

- The function being called (the callee)
- The function receiving the result (the caller)

Simple example: the function must only accept natural numbers (a prerequisite) and will always return natural numbers (a guarantee).

And just like in real life, these contracts can be violated.

Contract violations and blame assignment

When a contract violation occurs, blame must be assigned:

- Prerequisite violation → blame is on the caller.
- Guarantee violation → blame is on the callee.

Adding contracts to your code:

- Aids in debugging
- Provides automated runtime enforcement of constraints and invariants

We use the `typed-contracts` contract library by Hinze et al.

2.1 The `typed-contracts` library

Constructing a contract

`typed-contracts` uses a GADT:

```
data Contract a where
   Prop       :: (a → Bool) → Contract a
   Function   :: Contract a → (a → Contract b) →
        Contract (a ⇾ b)
   Pair       :: Contract a → (a → Contract b) →
        Contract (a, b)
   List       :: Contract a → Contract [a]
   Functor    :: Functor f ⇒ Contract a → Contract (f a)
   Bifunctor  :: Bifunctor f ⇒ Contract a → Contract b →
         Contract (f a b)
   And        :: Contract a → Contract a → Contract a
```

Constructing a contract - `Prop` constructor


```
Prop :: (a → Bool) → Contract a
```

- Lift a function to a contract
- Defines a constraint or property on a value

Constructing a contract - `Function` constructor §2.1

```
Function :: Contract a → (a → Contract b) → Contract (a
    ⇾ b)
```

- Defines a dependent function contract
- Note the ⇾

Constructing a contract - `Pair` constructor

```
Pair :: Contract a → (a → Contract b) → Contract (a, b)
```

- Defines a dependent pair
- Not used in this presentation

Constructing a contract - `List` constructor §2.1

```
List :: Contract a → Contract [a]
```

- Lifts contracts to the list level

Constructing a contract - `Functor` constructor

```
Functor :: Functor f ⇒ Contract a → Contract (f a)
```

- A container type that can house types of kind $* \rightarrow *$
- Examples: `Maybe`, `Just`

Constructing a contract - `Bifunctor` constructor§2.1

```
Bifunctor :: Bifunctor f ⇒ Contract a → Contract b →
    Contract (f a b)
```

- A container type that can house types of kind $* \to * \to *$
- Examples: `Either`, 2-tuple

Constructing a contract - `And` constructor


```
And :: Contract a → Contract a → Contract a
```

- Chains contracts together
- All contracts are asserted when a value is provided

Some notation for contracts


```
c1 -> c2 = Function c1 (const c2)
(&) = And
c1 <@> c2 = c1 & Functor c2
c1 <@@> (c2,c2) = c1 & Bifunctor c2 c3
```

- c_1 `->` c_2 defines a non-dependent function contract
- `<@>` and `<@@>` use c_1 as a contract that must hold on the container in its entirety: an outer contract.
- Example: an ordered list

Constructing a contract: examples

Fundamental contracts:

```
true, false :: Contract a
true  = Prop (λ_ → True)
false = Prop (λ_ → False)
```

A contract that only allows natural numbers:

```
nat :: Contract Int
nat = Prop (λi → i ≥ 0)
```

Asserting contracts

To attach a contract to a function, we use `assert`:

```
assert :: String → Contract a → a → a
```

`assert` acts as a partial identity function: in the case of a contract violation, an exception is thrown. Otherwise, it acts as identity.

Asserting contracts: an example


```
assert :: String → Contract a → a → a
```

```
inc :: Int ⇾ Int
inc = assert "inc" (nat ⇾ nat) (fun (λn → 1 + n))
```

- (nat ⇾ nat) is of type Contract (Int ⇾ Int)
- So, a must be of type (Int ⇾ Int)
- fun lifts a single argument to the contract level:

```
fun :: (a → b) → (a ⇾ b)
```

Asserting contracts: an example


```
inc :: Int ⇾ Int
inc = assert "inc" (nat ⇾ nat) (fun (λn → 1 + n))
```

We use app to apply values to a contracted function such as `inc`:

```
app :: (a ⇾ b) → Int → a → b
```

It also labels the application with a number, used in feedback:

```
> app inc 1 5
> 5
> app inc 1 (-5)
> *** Exception: contract failed: the expression
                 labeled '1' is to blame.
```

3. Contract inference

Inferring contracts?

- Jurriën Stutterheim describes a way to infer contracts for the components of a function in his thesis.
- Developed a contract inference algorithm: Algorithm $\mathcal{CW}$
- Based on Algorithm $\mathcal{W}$ by Damas and Milner
- Works on a small let-polymorphic lambda calculus

Three requirements for contract inference:

- Infer a well-typed contract for every component of a program
- Inferred contracts must allow a (non-strict) subset of the values allowed by the types
- The most general inferred contract must never fail an assertion

An intermediate contract grammar

$$
\begin{array}{lcl}
c & ::= & \rho_\alpha \\
 & | & true_\alpha \\
 & | & false_\alpha \\
 & | & c_\alpha \twoheadrightarrow c_\beta \\
 & | & c_\alpha \mathrel{<@>} c_\beta \\
 & | & c_\alpha \mathrel{<@@>} (c_\beta, c_\gamma) \\
 & & (\ldots) \\
 & & \\
\sigma & ::= & c \\
 & | & \forall true_\alpha . \sigma
\end{array}
$$

- Contract grammar is library-agnostic
- They must be translated to a contract library of choice
- Instead of fresh type variables, you have fresh *contract* variables

Examples of inferred contracts

- Function: `id :: a → a`
- Contract: $true_1 \multimap true_1$

- Function: `const :: a → b → a`
- Contract: $true_1 \multimap true_2 \multimap true_1$

- Function: `map :: (a → b) → [a] → [b]`
- Contract: $(true_1 \multimap true_2) \multimap (true_3 \, \langle @ \rangle \, true_1) \multimap (true_4 \, \langle @ \rangle \, true_2)$

Stutterheim's goal: superior feedback in Ask-Elle §3

- If a student's code does not follow a model solution, the only feedback possible is a QuickCheck counterexample
- Stutterheim wanted to express the QuickCheck properties as a contract for the main function
- Then use contract inference to infer contracts for the rest of the code
- Generate code that annotates all function applications with contract assertations
- Finally, apply the counterexample to the annotated code
- A contract violation occurs and offers a more precise location for the programming error

4. Expanding on Stutterheim's work

Limitations present in Stutterheim's system

We address:

- A system for code generation is left implicit
- Substitutions generated by Algorithm $\mathcal{CW}$ are placed in a global set, which may result in generating an inferred contract that causes a violation during assertion

We do *not* address:

- Inability of Algorithm $\mathcal{CW}$ to handle dependent contracts
- Lack of constant expression contracts
- Full integration with the Ask-Elle programming tutor

Our contributions

- We extend the contract inference algorithm to the Ask-Elle syntax, based on Helium, producing Algorithm $\mathcal{CHW}$
- Before performing contract inference, we perform AST transformations to simplify contract inference
- We generate *initial contracts* that simplify contract inference even further, especially in the case of mutually recursive functions
- Substitutions are divided into two lists: global and local, avoiding the aforementioned contract violation problem
- We provide a system to generate code for the *typed-contracts* library

4.1 System overview

4.2 AST transformations

4.3 Type source

4.4 Contract inference

4.5 Code generation

4.6 Generation of final contracts

5. Results

6. Future work

7. Conclusions

Questions?