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

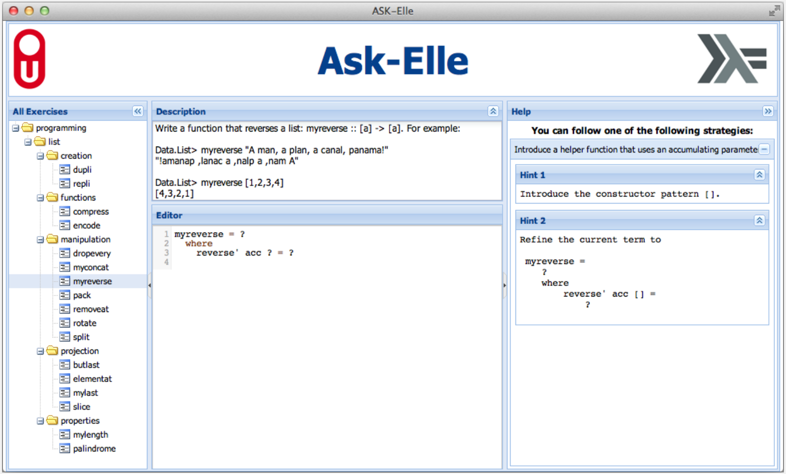
Ask-Elle?

- 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 §2.1

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

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

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 §2.1

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

- A container type that can house types of kind $* \to *$
- 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 §2.1

```
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
```

- c1 -▷ c2 defines a non-dependent function contract
- <@> and <@@> use c1 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

$$
\begin{array}{lcl}
c & ::= & \rho_\alpha \\
  & | & true_\alpha \\
  & | & false_\alpha \\
  & | & c_\alpha \rightarrow c_\beta \\
  & | & c_\alpha \; \langle @ \rangle \; c_\beta \\
  & | & c_\alpha \; \langle @@ \rangle \; (c_\beta, c_\gamma) \\
  & (\ldots) & \\
  & & \\
\sigma & ::= & c \\
  & | & \forall true_\alpha . \sigma
\end{array}
$$

- 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: $(true_1 \multimap true_2) \multimap (list_1 \lt@\gt true_1) \multimap (list_2 \lt@\gt true_2)$

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 assertions
- 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:

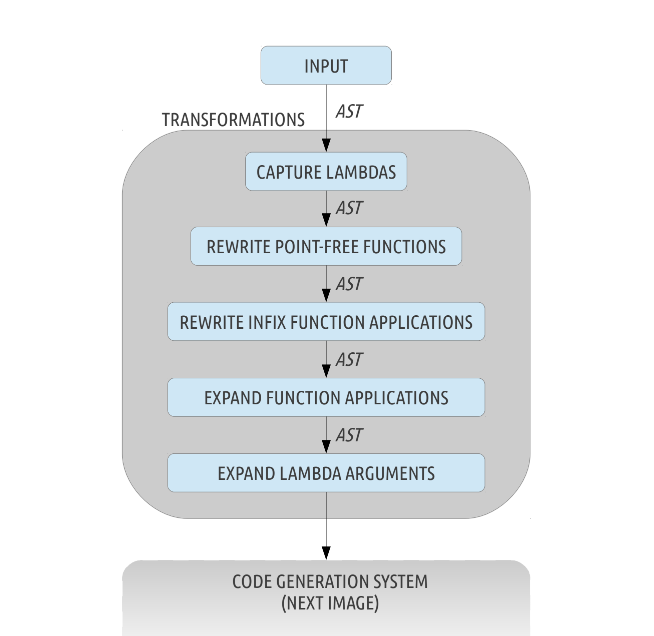
- Inability of Algorithm $\mathcal{CW}$ to handle dependent contracts
- Lack of constant expression contracts
- Full integration with the Ask-Elle programming tutor

Our contributions

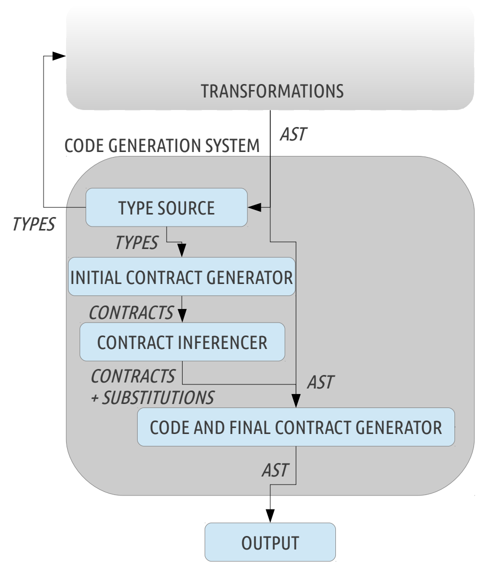
- We extend the contract inference algorithm to the Ask-Elle syntax, based on Helium, producing Algorithm $\mathcal{CHW}$
- Before performing contract inference, we perform AST transformations to simplify contract inference
- We generate *initial contracts* that simplify contract inference even further, 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 Framework overview

Framework overview - 1 / 2



Framework overview - 2 / 2



4.2 AST transformations

AST transformations - Capture lambdas

- Anonymous functions cannot be contracted: need a name
- Solution: bind all lambda functions to unique identifiers
- Referential transparency allows this

```
f = λx → x
```

is transformed into

```
f = __lam0
  where
    __lam0 = λx → x
```

AST transformations - Rewrite point-free functions

- `typed-contracts` does not support asserting partially applied functions
- all function arguments are made available on the LHS and applied to the RHS
- η-abstraction

```
f = __lam0
  where
    __lam0 = λx → x
```

is now transformed into

```
f __a0 = (__lam0) __a0
  where
    __lam0 __a0 = (λx → x) __a0
```

AST transformations - Rewrite infix function applications

§4.2

- Convert infix function applications to regular ones
- Prevent duplicate code for contract inference and code generation
- Removal of syntactic sugar

AST transformations - Expand function applications

- Split up function applications with multiple arguments into nested function applications
- Keeps the contract inference algorithm simple

```
f g x y z = g x y z
```

is transformed into

```
f g x y z = ((g x) y) z
```

AST transformations - Expand lambda arguments
§4.2

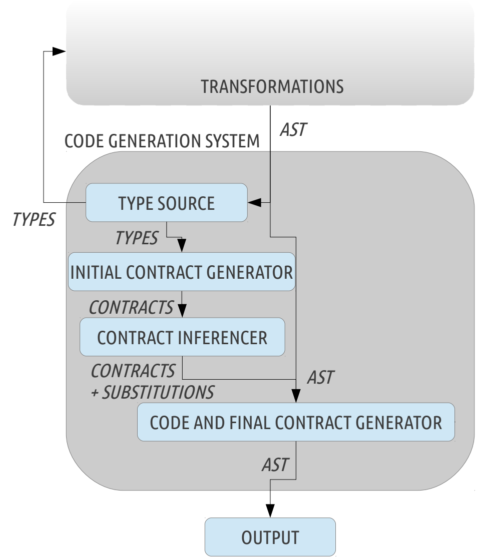
- Split up lambdas with multiple arguments into nested ones
- Again, keeps the contract inference algorithm simple

$$f = \lambda g\ x\ y\ z \rightarrow g\ x\ y\ z$$

is transformed into

$$f = \lambda g \rightarrow (\lambda x \rightarrow (\lambda y \rightarrow (\lambda z \rightarrow g\ x\ y\ z)))$$

Framework overview - Recap



4.3 Type source

What is a type source?

- A type source Ξ is a data structure that may hold information about the type of a node in an AST
- A node can query the type source for its type with $\Xi(x)$
- Types are available for the following AST nodes: `Expr`, `GuardedExpr`, `Pat`, `Alt`, `FunBind` and `Rhs`
- Type information is needed for some AST transformations
- Also useful for generating initial contracts

Initial contracts

- Used to simplify contract inference and code generation
- Captures relations between (mutually recursive) functions

Definition:

- Generalized version of the most specific contract for that identifier
- Asserting contract is equal to identity

Most-specific contracts

Stutterheim posits a conjecture:

- Any inferred contract in algorithm $\mathcal{CW}$ is also the most specific
- Contracts are represented as sets of values they allow
- Contract inferred for an expression e is a subset of any other contract that is valid (=acts as identity) for e

Generalized most-specific contracts

A generalized most-specific contract:

- Every contract variable maps to a $true_i$ contract (so no concrete contracts like `nat`)
- Relations present in an identifier or expression's type are also present in its contract
- We call them initial contracts because they seed the contract environment of Algorithm $\mathcal{CHW}$

Generalized most-specific contracts - Examples §4.3

Initial contracts for `id` and `map`:

```
ctrt_id  = c0 -> c0
ctrt_map = (c1 -> c2) -> (c3 <@> c1) -> (c4 <@> c2)
```

Not initial contracts:

```
ctrt_id  = c0 -> c1
ctrt_map = (c1 -> c2) -> (c3 <@> c4) -> (c5 <@> c6)
```

Why?

- `id`: relation between input and output is missing
- `map`: relation between function, list input and result missing

4.4 Contract inference

Algorithm $\mathcal{CHW}$

- Based on Algorithm $\mathcal{CW}$(so also on Algorithm $\mathcal{W}$)
- Works on Ask-Elle syntax
- Intermediate contract grammar is trimmed: $literal_i$
- Contract variables are implicitly universally quantified
- Definition of contract environment Γ is the same
- Use of type source Ξ to seed Γ
- Requirements remain the same (well-typed, subset of type inhabitants, inferred $=$ identity)

Relation between Γ and Ξ

- In type inference, Γ is updated with fresh type variables during inference
- These variables are later unified to infer the final type
- We need to infer contracts
- By taking initial contracts as starting point, we can focus on the inference uniquely required by contracts:

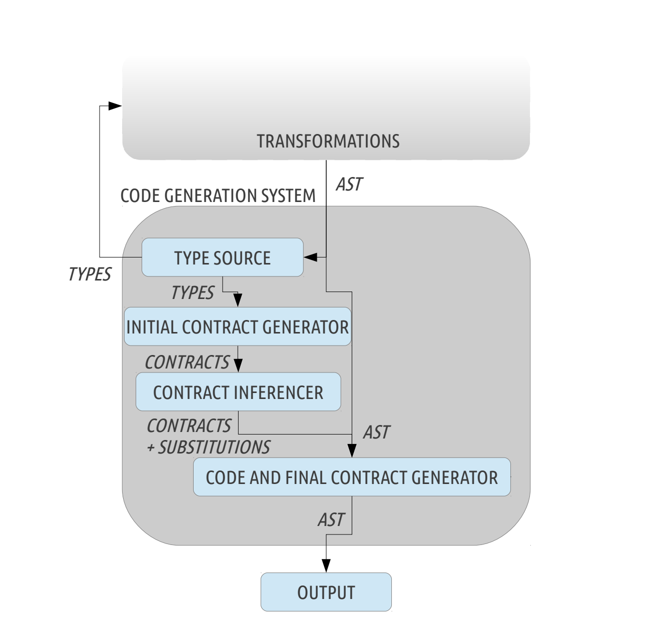
```
ctrt_map = (c1 -> c2) -> (c3 <@> c1) -> (c4 <@> c2)
```

- Perhaps c_3 and c_4 are the same?
- Cannot derive this from the type, need inference

Ξ does the heavy lifting

- Seed Γ with contracts for patterns and function identifiers
- When new identifiers come into scope, these are added as well, shadowing previous ones
- Described in seeding rules

Framework overview - Recap



4.5 Code generation

A code generation system

- Library-agnostic code generation system
- We can generate code for the `typed-contracts` library
- For each function `X`, we generate three new functions:
  - `___final_X`
  - `___app_X`
  - `___contracted_X`
- We will look at an example of each

___final_X

- Transformed copy of the original function definition
- Every function application is replaced by its contracted equivalent, the ___app_X template
- Adds position information from code for richer feedback
- Entry point for all other generated functions

__final_X - Example

```
f x = g x
g x = [x]
```

Generated code for `f x = g x`:

```
__final_f :: a → [a]
__final_f x = __app_g ctrt posg (posx,x)
```

Placeholders:

- *ctrt*: contract used to assert original function (g, in our case)
- *posg*: Tuple of line and column position of application of g
- *posx*: Tuple of line and column position of `x`

- Wraps arguments with a variation of `app` named `appParam`
- Passes them to the `___contracted_X` template
- A fixed template with placeholders that are filled in:

```
__app_X ctrt posinfo (argument patterns) = (applied arguments)
```

Placeholders:

- *(argument patterns)*: for each pattern `p` of function `X`, make a tuple `(posp,p)`
- *(applied arguments)*: wrap arguments with `appParam` and pass them to `___contracted_X` template

__app_X - Example

Generated code for function definition `f x = g x`:

```
type LC = Maybe (Int, Int)
__app_f :: Contract (a ⇀ [a]) → LC → (LC,a) → [a]
__app_f ctrt posinfo (posx,x) =
 appParam (__contracted_f ctrt posinfo) (show x ++
     generatePositionData posx) x
```

app takes an integer as a position label to use in feedback, `appParam` takes a string:

```
app :: (a ⇀ b) → Int → a → b
appParam :: (a ⇀ b) → String → a → b
```

__contracted_X

- Constructs an assertion with the contract provided
- A fixed template with placeholders that are filled in:

```
__contracted_X ctrt posinfo = assertPos (function info)
  (generatePositionData posinfo) ctrt funs
   where funs = (contracted function definition)
```

Placeholders:

- *function info*: String identifying the function responsible for the violation, for feedback purposes
- *contracted function definition*: Using `fun`, the function is lifted to a contracted version

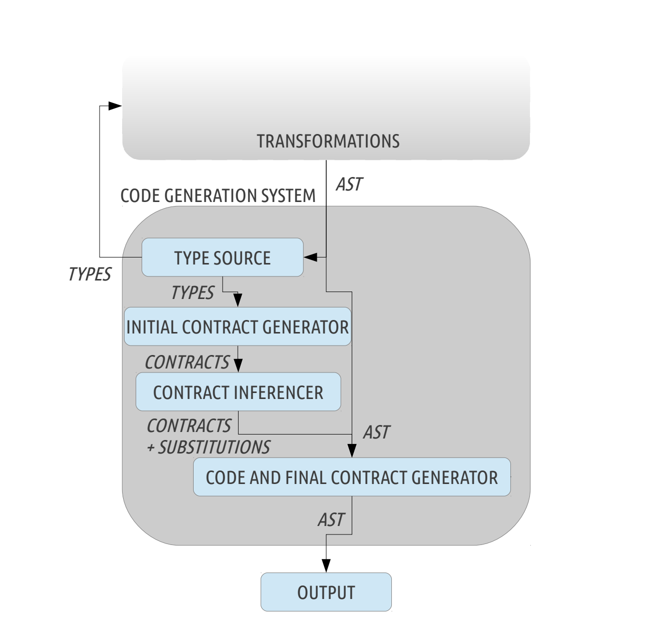
___contracted_X - Example

Generated code for function definition `f x = g x`:

```
__contracted_f :: Contract (a → b) → LC → (a → a)
__contracted_f ctrt posinfo = assertPos "f"
 (generatePositionData posinfo) ctrt funs
  where funs = fun (λx → __final_f x)
```

- `assertPos` takes position information for more precise feedback
- Remember that `___final_f` calls ___app_g
- For recursive functions: cycle of `___final_X` → ___app_X → `___contracted_X` → `___final_X` until base case is reached

Framework overview - Recap



4.6 Generation of final contracts

Contract translation and substitution application
§4.6

- Remember, inferred contracts are written in an intermediate contract language
- Translation to a target library required
- First, an example: `typed-contracts`
- Then: How substitutions are applied
- Translation occurs after substitutions

Translation to `typed-contracts`

Intermediate contract language is inspired by *typed-contracts*' notation, so translation is easy:

- $true_i \rightarrow$ `true`
- $false_i \rightarrow$ `false`
- $(\rightarrowtriangle)$, $(\langle @ \rangle)$ and $(\langle @@ \rangle)$ translate directly
- $literal_i \rightarrow$ `true` (future work)
- ρ_α: α is just a string that is printed during code generation

Substitutions in Stutterheim's proof-of-concept §4.6

- All substitutions are placed in a global set
- In some circumstances, may violate the "inferred = identity" requirement

The counterexample - 1 / 3


```
g x = [x]
z = (g 'a', g 5)
```

$$\Gamma(z) = \mathtt{true}_0 \ \langle@@\rangle \ (\mathtt{true}_1 \ \langle@\rangle \ \mathtt{isChar}, \ \mathtt{true}_2 \ \langle@\rangle \ \mathtt{isNum})$$

During inference:

$$\begin{aligned}
\Gamma(x) &= \mathtt{true}_4 \\
\Gamma(g) &= \mathtt{true}_4 \rightarrow (\mathtt{true}_3 \ \langle@\rangle \ \mathtt{true}_4) \\
\Gamma('a') &= \mathtt{true}_5 \\
\Gamma(5) &= \mathtt{true}_6
\end{aligned}$$

```
-- Unification of first argument of g with 'a'
unify (true₄) (true₅)

-- Unification of first argument of g with 5
unify (true₄) (true₆)
```

The counterexample - 2 / 3


```
-- Unification of first argument of g with 'a'
unify (true₄) (true₅)

-- Unification of first argument of g with 5
unify (true₄) (true₆)
```

Resulting substitutions: $\mathtt{true}_4 \mapsto \mathtt{true}_5$, $\mathtt{true}_4 \mapsto \mathtt{true}_6$

Apply substitutions to RHS of z, which has contract $(\mathtt{true}_7 \lt@@\gt (\mathtt{true}_3 \lt@\gt \mathtt{true}_4, \mathtt{true}_3 \lt@\gt \mathtt{true}_4))$:

- left-to-right: $(\mathtt{true}_7 \lt@@\gt (\mathtt{true}_3 \lt@\gt \mathtt{true}_5, \mathtt{true}_3 \lt@\gt \mathtt{true}_5))$
- right-to-left: $(\mathtt{true}_7 \lt@@\gt (\mathtt{true}_3 \lt@\gt \mathtt{true}_6, \mathtt{true}_3 \lt@\gt \mathtt{true}_6))$

The counterexample - 3 / 3

Unify with contract of $\mathtt{z}$:

```
unify (true_0 <@@> (true_1 <@> isChar, true_2 <@> isNum))
      (true_7 <@@> (true_3 <@> true_5, true_3 <@> true_5))
```

Relevant substitutions: $\mathtt{true}_5 \mapsto \mathtt{isChar}$, $\mathtt{true}_5 \mapsto \mathtt{isNum}$

```
unify (true_0 <@@> (true_1 <@> isChar, true_2 <@> isNum))
      (true_7 <@@> (true_3 <@> true_6, true_3 <@> true_6))
```

Relevant substitutions: $\mathtt{true}_6 \mapsto \mathtt{isChar}$, $\mathtt{true}_6 \mapsto \mathtt{isNum}$

Applying these substitutions always results in an incorrect contract:

- left-to-right: contract of $\mathtt{g}\ 5$ is $(\mathtt{true}_1\ \texttt{<@>}\ \mathtt{isChar})$
- right-to-left: contract of $\mathtt{g}$ $\mathtt{'a'}$ is $(\mathtt{true}_1\ \texttt{<@>}\ \mathtt{isNum})$

Solution: Global and local substitution lists

- Contract variables are given a flavour of global or local
- $true_i$ contracts are global
- Concrete contracts (like `nat`) are local
- Whenever unification takes place, split substitutions into two groups
- Global group: all substitutions from global $\rightarrow$ global
- Local group: all substitutions from global $\rightarrow$ local and local $\rightarrow$ local
- Global group is added to global list (up the AST)
- Local group is kept in the node where unification took place, also passed down to children (down the AST)

A snare in the plan!

- First, apply globals, then update locals, then apply locals
- Applying these substitutions statically presents a problem:

```
f x = g x
g x = [x]
z = (f 'a', f 5)
```

```
__final_f x = __app_g (c1 -> (c2 <@> c1)) x
  where c1 = true
        c2 = true
__final_g x = [x]
__final_z = (__app_f (isChar -> (true <@> isChar)) 'a',
     __app_f (isNum -> (true <@> isNum)) 5)
```

- The local substitutions that made the contract passed to __app_f more specific, are not utilized by __app_g
- Solution: Pass around and apply local substitutions at runtime

An even bigger snare in the plan!

- Substitutions only work on intermediate contract language
- But generated code expects contracts of targeted library!
- Runtime translation of intermediate contract language and target library is required
- `typed-contracts` contract is defined as a GADT with a phantom type for type-level representation of the contract
- Intermediate contract language is a UUAGC-generated regular data type
- Need a way to recover the extra type information
- Explored several avenues, left as future work

5. Results

Ask-Elle's feedback versus our framework's feedback

§5

- The feedback we generate is at least as informative as that of Ask-Elle
- Can't do much if code does not contain any function applications:

```
myreverse [] = []
myreverse [x,y] = [x,y]
myreverse xs = ?
```

- Automatically translating QuickCheck properties to contracts is often suboptimal, if doable at all:

```
prop_Main = λxs n → n > 0 ==> concatMap (replicate n
    ) xs == repli xs n
```

- Custom contracts offer better feedback capabilities

Ask-Elle's feedback versus our framework's feedback - Custom contracts


```
insert z zs = case zs of
                [] → [z]
                (z':zs') → in case z <= z' of
                                True → z' : z : zs'
                                False → z' : insert z zs'
foldr f b xs = case xs of
                [] → b
                (y:ys) → f y (foldr f b ys)

isort us = foldr insert [] us
```

The following QuickCheck property is checked, and fails:

```
prop_Main = λxs → whenFail (putStrLn "The list elements
    must be in ascending order.") (isOrdered xs)
```

QuickCheck reports a counterexample of `[0,1]`

Ask-Elle's feedback versus our framework's feedback - Custom contracts

Our custom contract:

```
isort_ctrt = (true <@> true) ↣ (ord <@> true)
 where ord = PropInfo (λx → isOrdered x) (λp →
     mkErrorMsg p "the list elements must be in ascending
      order.")
```

Code is generated, and we apply `___final_isort` to `[0,1]`...

Ask-Elle's feedback versus our framework's feedback - Custom contracts


```
1  insert z zs = case zs of
2                  [] → [z]
3                  (z':zs') → in case z <= z' of
4                                  True → z' : z : zs'
5                                  False → z' : insert z zs'
6  foldr f b xs = case xs of
7                  [] → b
8                  (y:ys) → f y (foldr f b ys)
9
10 isort us = foldr insert [] us
```

A part of your code, or a supplied argument to a function, does not fulfill a required property. This occurred at the application of the function 'insert' at line number 10, column number 19. The result of this function does not fulfill the following property: the list elements must be in ascending order.

Ask-Elle's feedback versus our framework's feedback - Custom contracts

- The issue lies with `insert`
- Cut down search space a little
- In bigger programs, search space will be cut down by a lot more

Ask-Elle's feedback versus our framework's feedback - Custom contracts

Let's rectify the error (it should be `z : z' : zs'`)...

```
insert z zs = case zs of
                [] → [z]
                (z':zs') → in case z <= z' of
                                True → z' : z : zs'
                                False → z' : insert z zs'
foldr f b xs = case xs of
                [] → b
                (y:ys) → f y (foldr f b ys)

isort us = foldr insert [] us
```

Ask-Elle's feedback versus our framework's feedback - Custom contracts

Let's rectify the error (it should be `z : z' : zs'`)...

```
insert z zs = case zs of
               [] → [z]
               (z':zs') → in case z <= z' of
                                True → z : z' : zs'
                                False → z' : insert z zs'
foldr f b xs = case xs of
               [] → b
               (y:ys) → f y (foldr f b ys)

isort us = foldr insert [] us
```

Ask-Elle's feedback versus our framework's feedback - Custom contracts

And introduce another by changing `z <= z'` to `z' <= z`:

```
insert z zs = case zs of
                []       → [z]
                (z':zs') → in case z <= z' of
                                  True  → z : z' : zs'
                                  False → z' : insert z zs'
foldr f b xs = case xs of
                []     → b
                (y:ys) → f y (foldr f b ys)

isort us = foldr insert [] us
```

Ask-Elle's feedback versus our framework's feedback - Custom contracts

And introduce another by changing `z <= z'` to `z' <= z`:

```
insert z zs = case zs of
                []       → [z]
                (z':zs') → in case z' <= z of
                                True  → z : z' : zs'
                                False → z' : insert z zs'
foldr f b xs = case xs of
                []     → b
                (y:ys) → f y (foldr f b ys)

isort us = foldr insert [] us
```

Ask-Elle's feedback versus our framework's feedback - Custom contracts

Let's apply the counterexample again...

```
insert z zs = case zs of
                []        → [z]
                (z':zs')  → in case z' <= z of
                                True  → z : z' : zs'
                                False → z' : insert z zs'
foldr f b xs = case xs of
                []        → b
                (y:ys)    → f y (foldr f b ys)

isort us = foldr insert [] us
```

Ask-Elle's feedback versus our framework's feedback - Custom contracts


```
1  insert z zs = case zs of
2                  [] → [z]
3                  (z':zs') → in case z' <= z of
4                                   True → z : z' : zs'
5                                   False → z' : insert z zs'
6  foldr f b xs = case xs of
7                  [] → b
8                  (y:ys) → f y (foldr f b ys)
9
10 isort us = foldr insert [] us
```

A part of your code, or a supplied argument to a function, does not fulfill a required property. This occurred at the application of the function ':' at line number 4, column number 45. The result of this function does not fulfill the following property: the list elements must be in ascending order.

Ask-Elle's feedback versus our framework's feedback - Custom contracts

- Something is wrong with `z' : zs'`: one of the elements is causing the list to no longer be ordered
- We cannot get closer to the problem: the contract of `(<=)` is `(true` $\rightarrow$ `(true` $\rightarrow$ `true))`
- Still, far more detailed feedback than Ask-Elle

6. Future work

Future work

- Full integration with Ask-Elle
  - Capture QuickCheck counterexamples and make them available to framework
  - Modify exercise configuration files to contain contracts
  - Run Prelude through the framework
- *Decoupling* from Ask-Elle: framework is only tightly bound to Helium for the type source
- Initial contract generation algorithm does not support user-defined data types
- Runtime translation of intermediate contract language
- Improve algorithm for finding final contracts
- Add support for partially applied function arguments to higher-order functions in code generation
- Use paramorphisms to tackle dependent contracts

7. Conclusions

Conclusions

Main objectives:

- Extend Stutterheim's contract inference algorithm to Ask-Elle syntax
- Construct a code generation system

Implementation:

- Constructed a framework coupling contract inference and code generation
- Algorithm $\mathcal{CHW}$ builds upon Algorithm $\mathcal{CW}$
- Library-agnostic code generation system, little code duplication required
- Example implementation for `typed-contracts` library
- Substitution problem was largely solved

Questions?