Universiteit Utrecht

Department of Computing Science

MSc Thesis, ICA-XXXXXXX

Contract Inference for the Ask-Elle Programming Tutor

Author:
Beerend Lauwers

Supervisor:
Prof. dr. J.T. Jeuring

April 8, 2014

Abstract

Ask-Elle [8] is a programming tutor that utilizes (transformations of) model solutions to provide feedback to students on their progress. If a student's program cannot be reduced to such a model solution, providing helpful feedback becomes hard. The Master Thesis of Jurriën Stutterheim [17] focuses on the development of a contract inference system for functional programs, with the goal to embed this functionality in the Ask-Elle programming tutor to provide a new source of meaningful feedback to students using it. This thesis builds upon his work, extending the contract inference algorithm to work with the language used by Ask-Elle: Helium. This language is a subset of the Haskell functional programming language. Additionally, a code generation system for the *typed-contracts* Haskell library is developed that annotates an arbitrary section of Helium code with contracts inferred by the extended contract inference algorithm. We provide some examples of the improved feedback of our system compared to the less-detailed feedback of Ask-Elle given the same inputs.

Contents

1 Introduction **4**

2 Background **6**

2.1 Contracts 6

2.1.1 The *typed-contracts* library 7

2.2 Contract inference 12

2.2.1 Goals of contract inference 12

2.2.2 Language used 13

2.2.3 Contract grammar 13

2.2.4 Contract relations and refinement 14

2.2.5 Algorithm $\mathcal{CW}$ 14

2.3 Related work concerning contracts 15

3 Contributions **18**

4 System design **20**

4.1 Syntax 20

4.2 Overview 21

4.3 AST transformations 23

4.3.1 Capture lambdas 23

4.3.2 Rewrite partially applied functions 24

4.3.3 Rewrite infix function applications 24

4.3.4 Expand function applications 24

4.3.5 Expand lambda arguments 25

4.4 Source of types and initial contracts 25

4.4.1 Type source 26

4.4.2 Initial contract generation 26

4.5 Contract inference 29

4.5.1 Formal description of the contract language 29

4.5.2 The contract inference system 31

4.6 Code Generation 43

4.6.1 ___final_X 45

4.6.2 ___app_X 47

4.6.3 ___contracted_X 47

4.7 Generation of final contracts 49

4.7.1 Translation of intermediate contract language to *typed-contracts* library 49

4.7.2 The monomorphic Algorithm $\mathcal{CW}$ 49

4.8 Providing richer feedback 54

4.8.1 Modifications to *typed-contracts* library 54

4.8.2 External library-agnostic feedback 56

4.8.3 Example of detailed feedback for *typed-contracts* library . 58

4.8.4 Reusing QuickCheck properties 59

5 Results and future work **61**

5.1 Results 61

5.1.1 Strengths and weaknesses 62

5.2 Integration with the Ask-Elle programming tutor 65

5.3 Adding support for user-defined datatypes to the initial contract generation algorithm 66

5.4 Runtime translation of intermediate contract language 66

5.4.1 Add a phantom type to the data type representing the intermediate contract language 67

5.4.2 Use a typeclass 67

5.4.3 Use a typeclass + type synonyms 68

5.4.4 Use a typeclass + newtypes 68

5.4.5 Code duplication 69

5.5 More efficient algorithm for finding final contracts 69

5.6 Adding support for partially applied function arguments to higher-order functions in *typed-contracts* code generation 70

5.7 Using paramorphisms to tackle dependent contracts 71

6 Conclusion **73**

Appendices **75**

A Modified Ask-Elle syntax **76**

Chapter 1

Introduction

The Ask-Elle [8] programming tutor is a web-based programming tutor that provides feedback on a student's progress in programming exercises in a language similar to Haskell, named Helium. To facilitate this, the system is given several solutions to an exercise when defining an exercise. It then uses strategies to reduce a student's input to one of these solutions, even identifying how many steps the input is removed from the solution. Ask-Elle can then provide hints and other forms of feedback to help along the student. Because it is possible that a student may come up with a solution that is not among the solutions provided to Ask-Elle, QuickCheck [2] properties, which describe the properties of a correct solution, can be defined for an exercise. For instance, one of the properties of a sorting function is that the sorted list must be a permutation of the original list: no elements may be removed or added during sorting.

To test the student's solution, inputs are randomly generated and applied to the proposed solution until an input causes a violation of the properties (called a *counter-example*) or a testing limit is reached. In the case of a failed test, Ask-Elle provides the student with the counter-example as feedback. Compared to the rich feedback when a solution strategy can be identified, this information requires the student to take the counterexample and step through his code to attempt locating the programming error.

The motivation behind the Master Thesis of Jurriën Stutterheim [17] is to fill this void in the system's feedback. To do so, he develops a contract inference system for functional programs, producing a contract inference algorithm for a simply-typed lambda calculus and proving the algorithm sound.

Stutterheim aimed to combine the QuickCheck properties with the contract inference algorithm to generate contracted code that, in the event of a contract

violation due to a counter-example generated by QuickCheck, would pinpoint exactly where the issue lies in a student's code. The integration of the contract inference system in the Ask-Elle programming tutor remained future work.

This thesis builds upon Stutterheim's work, extending the contract inference algorithm to work with the syntax Ask-Elle uses to represent a student's code. This syntax is a slightly leaner version of the full Helium syntax, itself a subset of the Haskell programming language. More precisely, Helium is a subset of the Haskell 98 language specification.

Additionally, a code generation system is developed for the *typed-contracts* library by Hinze et al.[10], a Haskell library to define and assert contracts on functions. This code generation system is largely library-agnostic and can be extended or adapted to generate code for different libraries.

In the following chapter, we provide the reader with a background on contracts in functional programming languages, using examples from the *typed-contracts* library to explain the concept. Furthermore, we quickly go through the results of Stutterheim's thesis to understand the foundations upon which this thesis is built. Related work in contracts and contract inference is discussed, as well. Chapter 3 provides an overview of our improvements upon Stutterheim's work. Chapter 4 follows the flow of a code module as it goes through the entire system, the main parts consisting of the contract inference and code generation subsystems. As we encounter each subsystem, we explain key concepts, definitions and how the system differs from Stutterheim's original work (if applicable). In Chapter 5, we discuss improvements that can be made to our contract inference algorithm and other future work. We conclude in Chapter 6.

Chapter 2

Background

In this chapter, we familiarize the reader with some concepts that are used in the rest of this thesis, the most important ones being *contracts* and the notion of *inferring* types and contracts.

This thesis concerns itself with *contract inference*. Let us break up this term and investigate each element thoroughly before moving on.

2.1 Contracts

In programming, a *contract* is very much like its real-world counterpart: it stipulates prerequisites and guarantees between two parties. In our case, the parties are the function being called (the *callee*) and the function receiving the result (the *caller*).

As a simple example, one could specify a contract that a function will only accept a natural number: a prerequisite. Likewise, it is possible to state that a function must also always return a natural number: a guarantee. By combining prerequisites and guarantees for arguments and results, respectively, it is possible to construct very specific contracts that are quite useful for tracking down bugs. Just like in real life, these contracts can be violated.

Checking for violations can be done during compile time or at runtime. If done at compile time, the compiler will produce an error message. If contract checking is done during runtime, the program will halt with an exception. In both cases, the message details which contract was violated at what position in the code.

When a contract violation happens, blame must be assigned. A prerequisite violation is to be blamed on the *caller*: it was this function that should have provided a valid argument. Likewise, a guarantee violation is to be blamed on the *callee*: there is a fault in the function being called that causes it to return an invalid result.

In other words, adding contracts to one's code adds an element of automated testing and aids in debugging any faults that may arise by being able to specify what exactly went wrong, and where.

This thesis builds upon the thesis of Stutterheim [17], who developed an algorithm for contract inference on a simple recursive let lambda language with support for holes. While Stutterheim's contract grammar is library-agnostic, he chose to generate code for the *typed-contracts* library by Hinze et al. [10]. Also of interest is the library by Chitil [1], which is based on the work of Hinze et al. This newer library preserves a program's lazy semantics, which the `typed-contracts` library does not.

2.1.1 The *typed-contracts* library

Let us examine the library for which code will be generated. Hinze et al. define the `Contract` type using 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
```

Let us go over each data constructor:

`Prop` is what one uses to actually define a constraint or property on a value. Given a function that expects a value of type `a` and which returns a Boolean, it produces a contract for something of type `a`. In other words, it lifts a function to a contract.

`Function` is used to define a dependent function contract, depicted as a tailed function arrow: `↣`.

`Pair` similarly defines a dependent pair contract.

`List` is straightforward: it lifts contracts to the list level.

`Functor` is a container that can house types of kind $* \to *$.

`Bifunctor` is the same as `Functor`, but takes types of kind $* \to * \to *$ into account.

`And` is used to chain several contracts together: all of them are asserted when a value is provided.

Now that we know how a `Contract` can be constructed, let us look at a few examples. The very first example is the one we used when first introducing the concept of contracts, namely a contract that constrains a function such that it demands a natural number as its input, and the guarantee that it will return a natural number, too:

```
natInNatOut :: Contract (Int ⇀ Int)
natInNatOut = nat ⇀ nat
```

We need a way to attach this contract to such a function, which is what the `assert` function is for:

```
increase :: Int ⇀ Int
increase = assert "increase" natInNatOut (fun (λn → 1 + n))
```

Note how the type of `increase` does not reflect the natural number constraints imposed by the contract, because the contract is asserted at runtime.

The `fun` function lifts a function to a contracted one, one argument at a time:

```
fun :: (a → b) → (a ⇀ b)
```

Conversely, the `app` function lifts function application to contracts:

```
app :: (a ⇀ b) → Int → a → b
```

The integer provided to `app` is used to tag the function application and is referred to during feedback.

By applying these functions, the contracted version of a function can replace its regular version without issue. In this way, the contract mechanism acts as a partial identity function, because it either raises an exception when a contract is violated, or it returns the asserted value unmodified. This behaviour is clearly reflected in the type of `assert`:

```
assert :: String → Contract a → (a → a)
```

From this type, we also see why functions must be lifted to the contract level before they can be asserted. For example, if `a` maps to `Int ⇀ Int`, as it does for the `increase` example, the `assert` function expects a contract of type `Contract (Int ⇀ Int)`, and a contracted function of type `Int ⇀ Int`.

To clearly illustrate how `assert` works, we leave out the first `String` argument, which is used in feedback. We also omit some code pertaining to location information, also used for feedback purposes.

```
assert (Prop p)          a   = if p a then a else error "contract failed"
assert (Function c₁ c₂) f    = (λx → (assert (c₂ x) . f) x) . assert c₁
assert (Pair c₁ c₂) (a₁,a₂) = (λa₁' →(a₁',assert (c₂ a₁') a₂)) (assert c₁ a₁)
assert (List c)          as  = map (assert c) as
assert (Functor f)       as  = fmap (assert f) as
assert (Bifunctor c₁ c₂) as = bimap (assert c₁) (assert c₂s) as
assert (And c₁ c₂)       a   = (assert c₂ . assert c₁) a
```

Let's go over each case to understand what happens:

`Prop` : When asserting a property, we apply the predicate to the supplied value. If the predicate holds, this function acts as the identity function. If it does not hold, however, an exception is raised. In the unabridged code, the exception message also includes the position of the violation in the source code.

`Function` : For a `Function` contract, we first assert c_1 on the input of the function f. If this assertion holds, we pass the value x to f, as can be seen in the lambda function. The result of fx is then used to assert the codomain of the function using c_2.

`Pair` : Similarly, the `Pair` contract first asserts the contract c_1 to the first element in the pair. Provided it succeeds, the first element is passed to c_2. The resulting contract $(c_2\ a_1')$ is then used to assert the second element of the pair.

`List`, `Functor`, `Bifunctor` : These are all quite similar. The contract to be asserted is mapped over the elements of a container.

`And` : This contract is used to combine two contracts, so asserting one implies that both contracts hold on the provided value.

Now that we know how contracts are constructed and asserted, let us look at some convenience functions that will be used throughout this thesis:

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

The first definition allows us to define a non-dependent function contract, and the second is some simple syntactical sugar. The last two definitions merit some more detail. Along with (a) contract(s) for the elements of a container, a contract for the container in its entirety is provided as well in the form of c_1. A simple example why this is useful, is the guarantee that a list is sorted when it is returned by a sorting function. Such a property cannot be captured by any contract for a container's elements, as these elements only know about themselves and not about other elements in the container. Therefore, we require container-wide contracts, which are also called "outer contracts". Conversely, contracts for a container's elements are referred to as "inner contracts".

With these functions, we can easily describe complex contracts. Let us look at an example. First, we define two fundamental contracts, namely `true` and `false`, which always succeed and always fail, respectively, regardless of their input:

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

Let us construct a few contracts for the `head` function that will not raise an exception. The very first contract is simply the `true` contract itself, which will take the entire `head` function as an argument. But the contract could be made more specific:

```
true ↣ true :: Contract ([a] ↣ a)
```

In this case, the `true` contract is asserted on both the input and output of the `head` function, but no longer on the function in its entirety. We can make it even more specific:

```
(true <@> true) ↣ true :: Contract ([a] ↣ a)
```

Using the functor contract <@>, the `true` contract is asserted on the input list in its entirety, as well as each of its elements, along with the output of the function.

Up until now, we've only used the `true` contract. But we can also use contract variables to express relations between the input and output of a function. A further refinement of the contract for `head` exemplifies this:

```
(c_outer <@> c_inner) ↣ c_inner :: Contract ([a] ↣ a)
```

It makes sense that the element that is extracted from the list still obeys the contract c_{inner}. With this knowledge, let us look at a higher-order example: `fmap`. By using the functor contract again, we can define a contract for `fmap`:

```
(c₁ ↣ c₂) ↣ (c₃ <@> c₁) ↣ (c₄ <@> c₂)
```

Note how the contract variables of the contract are very similar to the type variables of the type of `fmap`:

```
fmap :: (a → b) → f a → f b
```

The contract variables c_1 and c_2 correspond to type variables a and b, respectively. Generally, we cannot perform such a mapping over containers, so they are assigned different contract variables: c_3 and c_4. During a process called inference, it may turn out that these variables are equal, but this information is usually not deducible from the function's type.

To understand why, let us look at the following example:

```
listPlus, listId :: [Int] → [Int]
listPlus = map (+1)
listId = map id

atMostTen :: Contract Int
atMostTen = Prop (λn → n <= 10)

mapContract = (true ↣ true) ↣ (c₃ <@> true) ↣ (c₄ <@> true)
```

When asserting `listId [1..10]` with the contract `mapContract`, we can safely replace both c_3 (which must be valid for the input) and c_4 (which must be valid for the output) with `atMostTen` without causing a contract violation. If we do the same for `listPlus [1..10]`, however, we *will* cause a contract violation, as the resulting list will contain an integer larger than 10.

Section 2.2 explains the concept of inference and its use in refining contracts even further.

At the moment, we have learned how contracts are composed and how they can be attached to functions that are lifted using the `fun` function. We have also briefly encountered the `app` function, which lifts function application to contracts. Apart from applying something to a contracted function, this function provides crucial information for use in feedback:

```
app :: (a ↣ b) → Int → a → b
app f loc x = apply f (makeloc (App loc)) x
```

In order to be able to precisely blame the expression that caused a violation, `app` expects an integer to be used as a location label in the feedback.

Here is an example of a contract violation:

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

Because (-5) is not a natural number, it is the reason for the contract violation, and is blamed accordingly using the location label. In this thesis, we enrich this feedback by including position information and displaying the value responsible for the contract violation.

For brevity, we do not expand upon the inner workings of the blaming system. Details can be found in Hinze et al's paper [10].

2.2 Contract inference

In computer science, the notion of *inference* most often refers to the method of *type inference*, which automatically deduces types for expressions written in a programming language.

For functional programs, the Hindley-Milner type inference system described by Damas and Milner [4] is the most well-known. The thesis of Stutterheim [17] builds upon this system, producing an algorithm for deducing *contracts* from expressions written in a simple let-polymorphic lambda calculus.

2.2.1 Goals of contract inference

Stutterheim set out three goals for his contract inference system, shown in Figure 2.1.

- Infer a well-typed contract for every function in 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

(*Contract inference for Functional Programs*, page 12)

Figure 2.1: Stutterheim's goals for contract inference.

The first goal states that, given a well-typed program, the algorithm must infer a contract for every function and its components. The second goal is sensible: any values constrained by the contract of an expression should be a subset of the inhabitants of the type of that expression. Lastly, the third goal implies that asserting any inferred contract is equal to the identity function. In other

words, inferred contracts should never cause a contract violation.

2.2.2 Language used

The contract inference algorithm works on λ_c, a let-polymorphic lambda calculus shown in Figure 2.2.

```
expr   ::=  x                                  -- Variable
        |   λexpr → expr                       -- Lambda abstraction
        |   expr expr                          -- Application
        |   let expr = expr in expr            -- Let binding
        |   case expr of                       -- Case block
              { expr → expr (; expr → expr)* }
        |   const                              -- Constants
        |   expr : expr                        -- List cons constructor
        |   []                                 -- List nil constructor
        |   Just expr                          -- Maybe Just constructor
        |   Nothing                            -- Maybe Nothing constructor
        |   (expr , expr)                      -- Pair
        |   Left expr                          -- Either left constructor
        |   Right expr                         -- Either right constructor
        |   expr ⊕ expr                        -- Binary operation
        |   ?                                  -- Holes


const  ::=  n                                  -- Integers
        |   b                                  -- Booleans
        |   c                                  -- Characters
        |   s                                  -- Strings
```

Figure 2.2: The λ_c language.

2.2.3 Contract grammar

Contracts are generated in an intermediate language-agnostic grammar (Figure 2.3), which can be translated to library-specific contracts.

We have already seen some examples of user-defined contracts as well as the definitions of the `true` and `false` contracts in the *typed-contracts* library. But in this grammar, the contracts $true$, $false$ and ρ also have an index α that is used to differentiate between different instances of the same type of contract. This index serves the same purpose as fresh type variables in Algorithm $\mathcal{W}$. *"Generating a fresh contract"* is a commonly used phrase in this thesis that refers to a fresh indexed contract.

The syntax of the contracted function arrow ($\rightarrowtail$), functor ($<@>$) and bifunctor ($<@@>$) is the same as that of its inspiration, the *typed-contracts* library syntax.

```
-- Contracts
c ::= ρ_α                          -- User-defined concrete contract
    | true_α                       -- true contract
    | false_α                      -- false contract
    | c_α ⇀ c_β                    -- Function contracts
    | c_α <@> c_β                  -- Functor contracts
    | c_α <@@> (c_β, c_γ)          -- Bifunctor contracts
    | int_α                        -- Succeeds for all integers
    | bool_α                       -- Succeeds for all booleans
    | char_α                       -- Succeeds for all characters
    | string_α                     -- Succeeds for all strings
    | list_α                       -- Succeeds for all lists
    | either_α                     -- Succeeds for all Eithers
    | maybe_α                      -- Succeeds for all Maybes
    | pair_α                       -- Succeeds for all pairs
-- Contract schemes
σ ::= c                            -- Contract
    |   ∀true_α.σ                  -- Universal quantification for contract
                                   -- indices
```

Figure 2.3: Intermediate grammar for the contracts.

Stutterheim illustrates in his thesis that dependent function contracts are troublesome to use in contract inference, so these are omitted from the grammar.

Terminals for literals and several container data types are available as default contracts for the corresponding types.

Contract schemes are used to universally quantify over *true* contracts.

2.2.4 Contract relations and refinement

Before the algorithm itself is described, Stutterheim defines relations between contracts by "regarding contracts as sets of Haskell values" and using set-theoretic operations on these sets. He uses these relations to formally define the notion of *contract refinement*, where a contract that constrains a set of values is replaced by another contract that constrains a subset of those values. This concept is what drives the contract inference algorithm.

2.2.5 Algorithm $\mathcal{CW}$

The algorithm itself is very much alike Algorithm $\mathcal{W}$:

- A contract environment Γ is defined that maps variables to contracts.

- Use of generalization and instantiation to introduce and remove universally quantified *true* contracts, respectively.
- A modified version of Robinson's unification algorithm that checks for free *contracts* during the occurs check. It also ensures that substitutions always refine the contract.

Stutterheim proves that the algorithm is sound with respect to the contracting rules in his thesis. For brevity, we do not replicate the proofs, contracting rules, unification rules nor code for Algorithm $\mathcal{CW}$ here. These elements can be found in section 3.4 of Stutterheim's thesis [17].

2.3 Related work concerning contracts

The concept of contracts dates back to the Eiffel programming language [13] in 1988, which included it as a language feature. Many other languages support contracts, or have libraries available to achieve the same functionality.

In a functional setting, contracts are less popular, but there has been a lot of research in this field. Initial steps were taken by Findler and Felleisen [7], who defined a typed lambda calculus with support for contracts for higher-order functions in the Scheme language. For Haskell, Hinze et al. [10] developed a dynamic contract library that builds upon Findler and Felleisen's findings. A static approach was taken by Xu et al. [21, 22], allowing contracts to be checked using symbolic computation at compile-time. This technique is similar to that of refinement types, which we will cover in a later paragraph.

Degen et al. [5] provide an overview of contract implementation in lazy languages and their pitfalls, demonstrating how each form of dynamic contract monitoring (eager, semi-eager and lazy) is inherently flawed, either changing program behavior or ignoring contract violations. The article also touches upon Xu's work on static contract systems, which the authors argue is similar to the eager monitoring style they discuss in their article. Eager contract monitoring is preferred by the authors as all contracts defined in it are faithful: they will always evaluate to their actual value, which implies that, if a program terminates, all of the contracts within it will have evaluated to true.

Other researchers have explored enriching the type system to replace the functionality offered by contracts, producing what is commonly called "refinement types". A refinement type contains a predicate that is assumed to hold for the elements of the refined type, constraining the inhabitants of the type in the same way as a contract does.

Rondon et al. [15] have developed a set of data types called Logically Qualified Data Types, often shortened to Liquid Types. Using liquid types, they are able to embed a decidable subset of dependent typing functionality in a Hindley-Milner typing system. With only the HM types and a predefined set of logical qualifiers, their algorithm is able to infer liquid types, which are dependent types that consist of conjunctions of the aforementioned logical qualifiers. After inferring these liquid types, they can be resolved using an SMT solver. In the end, one ends up with the strongest constraint possible on the expression that can be generated with the provided set of logical qualifiers. Implementations of liquid typing are available for OCaml [15] and Haskell (using GHC) [16]. The Haskell implementation is able to provide feedback if type checking fails, indicating the position(s) in the source code where things have gone wrong. Additionally, a HTML file is generated of the processed source code annotated with the inferred types.

A similar approach was taken by Vytiniotis et al. [19], whose framework converts contracts written in Haskell into a simpler lambda calculus, which itself is translated into first-order logic formulae. Finally, these formulae are solved using an off-the-shelf theorem solver.

Terauchi [18] also builds upon the recent developments in refinement types to present a system that is able to infer dependent types for a subset of the OCaml language without external input. Instead of taking a user-provided set of formulas, the algorithm uses counterexample guided abstraction refinement (CEGAR) to iteratively refine a lattice of candidate dependent types. Counterexamples are parts of the program that are untypable with the current lattice of candidate types. The algorithm then attempts to type the program with all available types instead of the lattice. If typing succeeds, the new candidates are added to the lattice and a new counterexample is selected. If it fails, then the program is untypable. As the algorithm itself generates the set of candidates, users can be sure that if the program is untypable, it really is untypable. On the other hand, because the types are inferred automatically, they are not necessarily the strongest types available.

Ranjit et al. [11] take another route to attain refinement types: they first attain refinement type constraints using the implementation of Rondon et al. [15], and then translate these types to a first-order imperative program. If and only if the assertions of this imperative program hold, the higher-order program typechecks. This verification can be done using several abstract interpretation techniques (of which CEGAR is one), which have readily available implementations for imperative languages. The proof of safety of the imperative program translates to the solutions of the refinement type constraints. Thus, these can be used to annotate the original higher-order program, obtaining the refinement types.

Cousot et al. [3] have implemented automatic contract inference for method

extraction, a common refactoring technique. They provide implementations for C# and Visual Basic .NET. The inferred contract satisfies four requirements: (i) it is valid for the extracted method; (ii) the contract takes into account language and programmer assertions; (iii) the contract of the refactored code is as precise as the non-refactored code, and (iv) the contract is as general as possible. The authors use an iterative approximation algorithm to attain contracts that fulfill these requirements, and prove that an exact solution is uncomputable.

The articles of Dimoulas & Felleisen [6] and Greenberg et al. [9] provide an excellent overview of recent developments pertaining to contracts, as well as as detailed comparisons between the different libraries and frameworks. [6] also goes into greater detail of the semantics of contract satisfaction. Findler and Wadler [20] provide a unified view of several type enrichments, yielding a system they call evolutionary types. This system can be used to 'evolve' dynamically typed programs into statically typed ones and do the same for statically typed programs, 'evolving' them into programs with refinement types.

Another form of type enrichment that enforces constraints is dimensional typing. In short, a (usually numerical) value is given a dimension (time, length, mass, etc.). Either through static analysis and inference or through type-level programming, these constraints can be enforced, ensuring that nonsensical operations, such as adding a time value and a mass value, are caught at compile-time. Kennedy's thesis [12] explores this domain extensively for the ML language. A number of dimensional typing libraries are available for Haskell, most of which are defined using type-level programming or using type families. Some simply constrain the input and output of functions, while others perform automatic conversion between compatible types, for example between meters and inches.

Chapter 3

Contributions

The contract inference system devised by Stutterheim is a solid base upon which we can build to support a language such as Helium:

- The grammar to describe a contract is library-agnostic. It borrows several elements from the *typed-contracts* library by Hinze et al. [10], which it uses as the example library to describe its code generation procedure.
- Using simple set-theoretic operations, Stutterheim defines relations between contracts, which he uses to prove how a contract can be more general than another.
- The language used, λ_c, is a simple let-polymorphic lambda calculus.
- His contract inference algorithm $\mathcal{CW}$ is based upon Milner's Algorithm $\mathcal{W}$ and uses Robinson's unification algorithm, two well-understood algorithms.

There are a few limitations in Stutterheim's system that we address:

- A system for code generation is left implicit in Stutterheim's thesis. Feedback towards a user is also not discussed.
- Substitutions generated by algorithm $\mathcal{CW}$ are placed in a global set. Conflicting substitutions may result in generating an inferred contract that causes a violation during assertion.

The following limitations are *not* addressed by us:

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

We improve upon Stutterheim's system in the following ways:

- The language we use is a slightly simplified version of the Helium language, a dialect of Haskell that is a subset of the Haskell 98 language specification.
- Using type information, we perform abstract syntax tree (AST) transformations to simplify the contract inference algorithm.
- With the same information, we generate *initial contracts* that simplify contract inference even further, especially in the case of mutually recursive functions.
- Algorithm $\mathcal{CW}$ is extended to support the aforementioned Helium dialect, producing Algorithm $\mathcal{CHW}$.
- Substitutions are divided into two lists: bottom-up and top-down. This avoids the problem of contract violations by inferred contracts inherent to the original system.
- We provide a system to generate code for the *typed-contracts* library and identify several library-agnostic techniques to enrich feedback.

Furthermore, we modify the *typed-contracts* library to allow for a richer form of feedback.

Chapter 4

System design

In this chapter, we explain the improvements we have made to Stutterheim's system in detail. The chapter follows a similar structure to the path code is put through as illustrated in section 4.2.

Before providing an overview of the system, we introduce the syntax for our abstract syntax tree (AST), which is a slightly modified version of the syntax the Ask-Elle programming tutor uses. After the overview, we follow the flow of the abstract syntax tree as it goes through the entire system, starting with several AST transformations, after which we enter the code generation system, which consists of contract generation and -inference, and code generation. As we encounter each subsystem, we explain key concepts and definitions.

4.1 Syntax

The Ask-Elle programming tutor uses its own version of the Helium syntax, casting aside some things such as type signatures and other syntax constructs that are irrelevant to Ask-Elle. Helium itself is a subset of the Haskell 98 language specification, its main missing component being user-definable type classes.

We have slightly modified the syntax used by the Ask-Elle system:

- Fields to store range information were added to nearly all nodes. Range information consists of the line and column number of the node and its source (standard input, file, unknown) if this information is available.

- In Ask-Elle, constructors that use a list of patterns or expressions use the `Pats` and `Exprs` type synonyms, respectively. For example, expressions that describe a tuple or a list both use `Exprs`. However, from the viewpoint of contract inference, these expressions could not be more different! The solution is simple: tuples and lists in patterns and expressions use the type synonyms `ListPatsR`, `TuplePatsR`, `ListExprR` and `TupleExprR`. All other constructors remain unchanged.

The structure of this modified syntax can be found under Appendix A. For an overview of the visual representation of the syntax, please refer to the Helium documentation, as code is first lexed and parsed into Helium syntax before being downcast to the Ask-Elle syntax.

The syntax can be downcast to the regular Ask-Elle syntax, and partially upcast to the Helium syntax.

In the rest of this thesis, we omit the `R` from the syntax for aesthetic reasons.

4.2 Overview

The system first passes the input through six AST transformations, which help simplify contract inference and code generation:

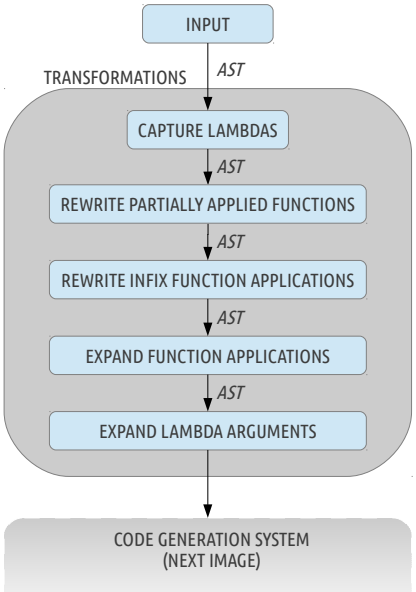

Figure 4.1: An overview of the applied AST transformations.

Afterwards, the AST is passed to two subsystems: the **type source** and the **code and final contract generator**.

The **type source** extracts types from the AST, passing them to the **initial contract generator**, which generates a set of contracts from the types. These contracts are then passed to the **contract inferencer**, which infers a set of contracts and a set of substitutions. These are passed to the **code and final contract generator**.

The **code and final contract generator** inspects the AST and generates a new version of the AST that incorporates contract assertions, producing a contract-annotated (or 'contracted') version of the original AST.

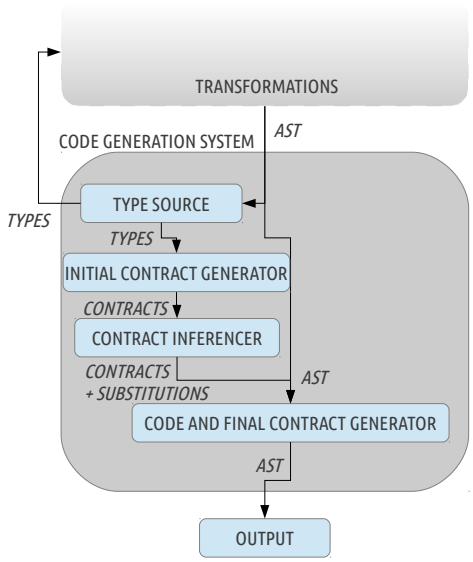

Figure 4.2: An overview of the steps required to output contract-annotated code.

4.3 AST transformations

Before passing the AST to the rest of the system, we apply several transformations to it to facilitate contract inference and code generation.

4.3.1 Capture lambdas

Anonymous functions cannot be contracted, as a name is required to properly perform generation of contracted code. Our solution is simple: bind all lambda functions to a unique identifier and place them in the where-clause of the original definition. A simple example:

```
f = λx → x
```

is transformed into

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

Listing 4.1: Result of lambda capture transformation.

Because of referential transparency, the behaviour of the program is not altered.

4.3.2 Rewrite partially applied functions

The library of our choice, *typed-contracts*, does not support asserting partially applied functions. Because of this, all function arguments are made available on the left-hand side and applied to the right-hand side of a definition. To continue with our previous example,

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

is now transformed into

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

Listing 4.2: Result of η-expansion transformation.

This is essentially η-expansion. The information required for this transformation is provided by the type source, which is described in section 4.4.

4.3.3 Rewrite infix function applications

To reduce duplicate code for both contract inference and code generation, we convert infix function applications to regular function applications. In other words, we remove a layer of syntactic sugar.

4.3.4 Expand function applications

To keep contract inference simple, we split up function applications that are applied to multiple arguments into several nested function applications. For example:

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

is transformed into

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

Listing 4.3: Result of function application expansion transformation.

Our syntax is parsed by the Helium compiler, which stores function applications with multiple arguments in a single node (`App g [x,y,z]`) instead of nested function application nodes (`App (App (App g [x]) [y]) [z]`. This decision to use a single node would require us to change Stutterheim's original contract inference code. By placing parentheses, we force the parser to produce nested function application nodes, allowing us to retain Stutterheim's code for function applications.

4.3.5 Expand lambda arguments

The reasons for this transformation are similar as the previous one: keeping contract inference simple and being able to reuse Stutterheim's contract inference code. A lambda function with multiple arguments is split up into nested lambda functions:

```
f = λg x y z → g x y z
```

is transformed into

```
f = λg → (λx → (λy → (λz → g x y z)))
```

Listing 4.4: Result of lambda argument expansion transformation.

Because this is done after the lambda function capture transformation, this transformation solely occurs in where-clauses, where the original lambda functions are bound to an identifier.

4.4 Source of types and initial contracts

After these transformations, the AST is used to generate a type source for the AST. The AST is also passed to the code generator. in this section, we explain the concept and role of a type source in the contract inference algorithm and code generation system.

4.4.1 Type source

Abstractly, a type source Ξ (pronounced "Xi") is a data structure that may hold information about the type of a node in an AST. A node can query the type source to obtain its type if it is available using the following notation: $\Xi(x)$.

Our particular type source consists of a doubly linked tree that resembles the corresponding AST. This "type tree" is generated by applying a modified version of the Helium type inference code to the AST, which traverses the AST and produces type information for the following AST nodes:

- `Expr`
- `Pat`
- `Alt`
- `FunBind`
- `Rhs`
- `GuardedExpr`

4.4.2 Initial contract generation

Using the type information present in the type source, we generate *initial contracts* that prove useful in simplifying contract inference and final contract generation.

Definition of initial contracts

By deriving an initial contract from an identifier's type, we obtain a contract with the following properties:

- Asserting the contract is equal to the identity function.
- It is a *generalized* version of the *most specific* contract for that identifier.

Let us discuss the latter property in further detail: Stutterheim posits a conjecture that any inferred contract in algorithm $\mathcal{CW}$ is also the *most specific*, which means that the contract inferred for an expression e is a subset of any other

contract that is valid (that is, it does not cause a violation when asserted) for e.

A *generalized* version of such a contract is one where the relations between type variables present in an identifier or expression's type are also present in the contract variables of its contract.

For example, here are generalized most specific contracts for the functions `id` and `map`:

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

Conversely, these are **not** generalized most specific contracts for those functions, as they do not capture the relations present in the types of the functions:

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

Generalized most specific contracts, also called *initial* contracts, are not unique: it is possible to have the initial contracts $\boxed{c_0 \rightarrowtail c_0}$ and $\boxed{c_1 \rightarrowtail c_1}$ for the identity function.

Uses for initial contracts

There are a few uses for initial contracts:

- The contract environment utilized by the inference algorithm is seeded with contracts from the type source before inference initiates. This is where the term *initial contract* comes from.
- They simplify contract inference by already incorporating some of the relations that would otherwise be discovered during inference. Particularly, relations between mutually recursive functions are captured as well.
- They allow for polymorphic versions of the *___final_*$\boxed{X}$, *___app_*$\boxed{X}$ and *___contracted_*$\boxed{X}$ templates.

The last two uses merit more detail, but to understand them, knowledge about the contract inference and code generation systems is necessary. The relation between the type source and the contract environment is explained in subsection 4.5.2. Explanation of the last use case is deferred to section 4.7.

Transferring relations between mutually recursive functions

Mutually recursive functions are troublesome to define in Stutterheim's λ_c, requiring the contract environment Γ to be pre-populated with the contract for the last-defined mutually recursive function to ensure Algorithm $\mathcal{CW}$ does not terminate with an error. In the case that Algorithm $\mathcal{CW}$ terminates without error, the relation(s) between the mutually recursive functions is recorded in their contracts.

Because initial contracts are derived from types that record any relation(s) functions may have, the equivalent of pre-populating Γ, binding group analysis, is no longer necessary. Binding group analysis is done during type inference to increase polymorphism in those types, an attribute we wish to mirror in our contracts. By piggybacking upon already-inferred types, we gain both this increased polymorphism and any relations between functions present in the types.

Initial contract generation algorithm

An initial contract is generated by passing the type representation (a data type from the *top* library) to the `convertTopTypeToContract` function, which strips down the type to its primitive type and converts it. In the inference rules, this conversion is called with the notation $\Xi_{ctrt}(x) = c$, x being the AST node that is querying for its type t and c being the result of `convertTopTypeToContract` applied to t.

The following list shows which pattern match relates to which action:

`(TVar i)` : A `TVar` contains an integer `i` that corresponds to a specific type. That integer is prefixed with the character 'c' to create a fresh contract variable. For example, if `a` maps to the integer `1`, the type `a` is converted to the contract `c1`.

`(TApp (TApp (TCon "→") t1) t2)` : A function application. `t1` and `t2` are also converted and put in a non-dependent function contract. For example, given the mapping `a ↦ 1, b ↦ 2` the type `a → b` is converted to the contract `c1 ↣ c2`.

`(TApp (TCon x) t1) | x `elem` ["Maybe","[]"]` : Both the `Maybe` type and the list type are converted to a functor contract.

`(TApp (TApp (TCon x) t1) t2) | x `elem` ["Either","(,)"]` : Both the `Either` type and the 2-tuple type are converted to a bifunctor contract.

`(TCon x) | x `elem` ["Int","Char","Float","Bool","String"]`
: Literals of these types are converted to a fresh contract variable that does not collide with those used for the `TVar` constructors.

3-tuples and 4-tuples are also supported, but have been left out for brevity.

At the moment, user-defined data types are unsupported. Implementing support is discussed in section 5.3 of the future work chapter.

4.5 Contract inference

Our contract inference system is based upon Stutterheim's work, and it is advised you read his chapters on contract inference if you are interested in the details, which we may omit if they are the same.

Our algorithm is also based on the Damas and Milner type inference system, with a few modifications and simplifications thanks to our use of a type source.

Our goals for contract inference remain the same as Stutterheim's:

1. Infer a well-typed contract for each function in a program;
2. Inferred contracts must allow a (possibly non-strict) subset of the values that the type allows, and
3. The most general contract inferred by the inference system must never fail during assertion.

Because we use Algorithm $\mathcal{CW}$ as a starting point, our algorithm, too, is based on Milner's Algorithm $\mathcal{W}$, and we make use of Robinson's unification algorithm as well. The resulting algorithm is called Algorithm $\mathcal{CHW}$.

4.5.1 Formal description of the contract language

Figure 4.3 describes our intermediate contract language, which is a subset of that of Stutterheim's contract language.

While Stutterheim preferred to have specific terminals for data types and literals, we feel this over-encumbers the contract language. Instead, we replace this multitude of terminals with a single $literal_i$ terminal.

```
-- Contracts
c ::= ρ_α                          -- User-defined concrete contract
    | c_α ↣ c_β                    -- Function contracts
    | c_α <@> c_β                  -- Functor contracts
    | c_α <@@> (c_β, c_γ)          -- Bifunctor contracts
    | true_α                       -- true contract
    | false_α                      -- false contract
    | literal_α                    -- literal contract
```

Figure 4.3: Simplified grammar for the contract language.

Furthermore, contract schemes for universal quantification of contract indices are removed from the grammar. Instead, every generated contract is implicitly universally quantified. Instantiation can be said to be deferred to substitution application due to the way we divide and apply substitutions as described in section 4.7.

Stutterheim describes several definitions, properties and relations between contracts in his thesis. The following of those can be brought over to our system without change:

Definitions:

- Definition 1. Equivalency of contracts.
- Definition 2. Semantics of contracts.

Propositions:

- Proposition 1. Contract relations with $true$ and $false$.
- Proposition 2. Assertion fails for subset.
- Proposition 3. Assertion succeeds for superset.
- Proposition 4. Superset relation for function contract semantics.

Propositions 5 (*Soundness of inference*) and 6 (*Asserting inferred contract is identity*) must be redone due to the new syntax, although the majority of the proofs for the `Expr` node will be very similar to Stutterheim's proofs. We leave these proofs as future work.

4.5.2 The contract inference system

In this section, we introduce a system that allows us to infer a contract from an expression. This system is based upon Stutterheim's work, and this section will roughly follow the same structure as the corresponding section in his thesis.

Contract environment

In our system, we use a contract environment Γ (pronounced "Gamma") that maps contract variables to contracts.

A contract environment is defined as such:

$\Gamma ::= [\,] \,||\, \Gamma_1[x \mapsto c]$

So, Γ can either be completely empty, or consist of an environment Γ_1 that is extended by a mapping from an identifier x to a contract c.

We use the same notation as Stutterheim for contract environments:

- $\Gamma(x) = c$ means that the right-most binding for x in Γ maps x to c.
- $\Gamma \vdash e : c$ denotes that, in environment Γ, the expression e has the contract c.
- $fc(\sigma)$ indicates the set of *true* contracts that are free in contract schema σ.
- $fc(\Gamma)$ indicates the set of *true* contracts that are free in the codomain of Γ.

The `gen` and `inst` support functions used by Stutterheim are depreciated because of our choice to implicitly universally quantify contract variables and by our use of a type source.

Before moving on to the contracting rules and Algorithm $\mathcal{CHW}$, we first explain in detail the relations and differences between the contract environment Γ and type source Ξ.

Relation between Ξ and Γ

In type inference, a type environment Γ is updated during the inference process. For instance, encountering a *let*-binding will add the bound identifier x to Γ so

it can be used in the rest of the *let*-binding. A similar process occurs for a lambda function's pattern(s). Fresh type variables are used when introducing these identifiers into Γ. These variables are later unified with others to infer the final type of an expression.

But in our case, we require contracts, not types, to be inferred. We have seen in subsection 4.4.2 how we can derive a generalized, most-specific contract from a type. What if we take these derived contracts and use them as a starting point from which to initialize inference? Would inference even be necessary anymore?

The answer is *yes*. When deriving a contract from a type, some information cannot be gleaned from the type: container types, such as those of lists, require a fresh contract to be used as their outer contract. For example, the relations between the outer contracts of an input list and an output list cannot be derived from inspecting the type. One must inspect the actual expression to which the type belongs.

Using contracts derived from types, then, allows us to focus on the inference uniquely required by (our definition of) a contract. For this reason, we seed the contract environment Γ with initial contracts from the type source Ξ for the following identifiers:

- Patterns
- Function identifiers

Seeding the contract environment Γ is not done once: it is seeded every time new identifiers come into scope, shadowing existing ones. The seeding rules that precisely describe this behaviour are very similar to the contracting rules of Stutterheim and inference rules of Damas and Milner.

Seeding rules

We use the following notation for the type source:

- $\Xi_{ctrt}(x) = c$ means that the type source is queried for the type of node x. This type is converted to an initial contract, the result being returned as c.

We begin with the seeding rules for patterns, followed by those for declarations.

Patterns. In Figure 4.4, the seeding rule `S-PVar` we see how the contract for a pattern variable is requested from the type source Ξ and added to the environment Γ. The same occurs for a pattern alias in `S-PAs`.

$$\frac{\Xi_{ctrt}(p) = c}{\Gamma[p \mapsto c]}\ \text{S-PVar}$$

$$\frac{\Xi_{ctrt}(p) = c \qquad \Xi_{ctrt}(ps) = c}{\Gamma[p \mapsto c]}\ \text{S-PAs}$$

Figure 4.4: Seeding rules for patterns.

Declarations. The seeding rules for declarations (Figure 4.5) add function identifier contracts to the environment(s) for the function binding(s). `S-DFunBinds` illustrates how the contracts derived from a declaration group must all be the same: `f (x:xs) = [x]` should have the same initial contract as `f [] = []`, for example. Relaxing this requirement would require some pattern matching analysis to ensure the correct contract is used during contract inference and code generation. At the moment, the initial contract of the first declaration in the group is used as the contract for the entire group.

For pattern bindings, the pattern's contract is requested from the type source and inserted into Γ.

$$\frac{\forall i \in [0 \ldots n]\ d_i = f\ p_i = rhs_i \qquad \Xi_{ctrt}(d_i) = c}{\Gamma[f \mapsto c]}\ \text{S-DFunBinds}$$

$$\frac{\Xi_{ctrt}(p) = c}{\Gamma[p \mapsto c]}\ \text{S-DPatBind}$$

Figure 4.5: Seeding rules for declarations.

Γ versus Ξ

Even though contract environment Γ has been seeded, it is sometimes required to directly query the type source Ξ using Ξ_{ctrt}.

For the seeding rules, we slightly simplified the notation for querying a type source. In reality, the type source is also given as many fresh contracts as is required to convert the type into a contract. These fresh contracts are exclusively used for outer contracts in functor and bifunctor contracts. This means that two queries to Ξ for the list constructor `(:[])`, for example, result in two different

contracts being delivered, the contracts differing in their fresh outer contracts. Contracting rules such as `C-PCon` and `C-Con` make use of this functionality.

The key difference between Γ and Ξ is this: contracts from Γ are *static* and are only changed by substitutions. Contracts from Ξ can be *dynamic*.

Now that it is clear what the relations and differences between Γ and Ξ are, we are ready to proceed to the contracting rules.

Contracting rules

We first introduce the contracting rules for patterns, as many of the rules are similar to those for declarations, right-hand sides, (guarded) expressions and alternatives.

$$\frac{fresh(i)}{\Gamma \vdash ? :: true_i}\ \text{C-PHole}$$

$$\frac{fresh(i)}{\Gamma \vdash * :: true_i}\ \text{C-PWildcard}$$

$$\frac{\Xi_{ctrt}(p_1) :: c_1 \rightarrowtail c \qquad \Gamma \vdash p_2 :: c_1}{\Gamma \vdash p_1\ p_2 :: c}\ \text{C-PCon}$$

$$\frac{\Gamma(p) = c}{\Gamma \vdash p :: c}\ \text{C-PVar}$$

$$\frac{\Gamma(p) = c}{\Gamma \vdash p@(ps) :: c}\ \text{C-PAs}$$

$$\frac{\Gamma \vdash x :: c \qquad \Gamma \vdash xs :: list \lessdot @ \gtrdot c}{\Gamma \vdash (x : xs) :: list \lessdot @ \gtrdot c}\ \text{C-PCons}$$

$$\frac{fresh(i, j)}{\Gamma \vdash [] :: true_i \lessdot @ \gtrdot true_j}\ \text{C-PNil}$$

$$\frac{\forall i \in [0 \ldots n] \Gamma \vdash e_i :: c_i \qquad fresh(j)}{\Gamma \vdash (e_0, .., e_i) :: true_j \lessdot @@ \gtrdot (c_0, .., c_i)}\ \text{C-PTuple}$$

$$\frac{\text{l is a literal} \qquad fresh(i)}{\Gamma \vdash l :: literal_i}\ \text{C-PLit}$$

$$\frac{\Gamma \vdash e_1 :: c_1 \qquad \Gamma \vdash e_2 :: c_2 \qquad \Xi_{ctrt}(\oplus) = c_1 \rightarrowtail c_2 \rightarrowtail c_3}{\Gamma \vdash e_1 \oplus e_2 :: c_3}\ \text{C-PInfixCon}$$

Figure 4.6: Contracting rules for patterns.

Patterns. Let us go over the rules in Figure 4.6. We omit `C-PParen` because of its simplicity. Please note that while our syntax contains a `PListR` data constructor, we use the two contracting rules `C-PCons` and `C-PNil` to describe its contracting behaviour. This is how the `PListR` constructor works behind the scenes. We will follow this convention for all constructors that describe lists.

`C-PHole` and `C-PWildcard` : In both cases, a fresh contract is generated because a more specific contract cannot be provided.

`C-PCon` : The contract $c_1 \rightarrowtail c$ is queried from Ξ. c may contain a fresh outer contract.

`C-PVar` and `C-PAs` : The contract for p is fetched from Γ. In the case of `C-PAs`, c is used for the entire pattern.

`C-PNil` : Because we know nothing about the inner or outer contract, we generate fresh *true* contracts for both.

`C-PCons` : In this case, we know about the contracts of the tail and head of the list. The inner contract of the list tail must be the same as the contract of the head of the list. The resulting contract is the same as the list tail.

`C-PTuple` : For a tuple; we know about each contract of the tuple's members. We generate a nested bifunctor contract for tuples larger than 2, nesting in c_2 until we have accommodated every member of the tuple. For each bifunctor contract, we require a fresh contract for the outer contract, which means we require $n - 1$ fresh contracts, n being the size of the tuple.

`C-PLit` : A pattern literal is given a fresh *literal* contract.

`C-PInfixCon` : An infix operator in Helium supports two arguments, so the contracting rule is a slight variation `C-PCon`.

$$\frac{fresh(i)}{\Gamma \vdash ? :: true_i}\ \text{C-DHole}$$

$$\frac{fresh(i)}{\Gamma \vdash :: true_i}\ \text{C-DEmpty}$$

$$\frac{\Gamma \vdash d :: c}{\Gamma \vdash (d : ds) :: c}\ \text{C-DFunBinds}$$

$$\frac{\Gamma(p) = c}{\Gamma \vdash p = rhs :: c}\ \text{C-DPatBind}$$

Figure 4.7: Contracting rules for declarations.

Declarations. Figure 4.7 shows the contracting rules for declarations. The rules `C-DHole` and `C-DEmpty` are trivial, but the remaining two warrant a little more explanation:

`C-DFunBinds` : In this rule, we select the contract of the very first element of the list of function bindings and use that as the contract of the entire list. This makes sense because function bindings must have the same number of arguments and the same result, meaning that the contract is also the same.

`C-DPatBind` : A pattern binding will look up its contract in the contract environment, in the same manner as `C-PVar`.

$$\frac{fresh(i)}{\Gamma \vdash ? :: true_i}\ \text{C-FBHole}$$

$$\frac{\Gamma(ident) = c}{\Gamma \vdash ident\ ps = rhs :: c}\ \text{C-FunBind}$$

$$\frac{\Gamma \vdash f :: c \qquad \Gamma \vdash fs :: c}{\Gamma \vdash (f : fs) :: c}\ \text{C-FCons}$$

Figure 4.8: Contracting rules for function bindings.

Function bindings. The contracting rules for function bindings in Figure 4.8 are straightforward: `C-FBHole` works as all other holes, and `C-FunBind` looks up the identifier in the contract environment and assigns that contract to the entire function binding. `C-FCons` illustrates that every element in a list of function bindings must have the same contract. The rule `C-FNil` does not exist, as it would be represented by a `DEmpty` declaration node.

$$\frac{\Gamma \vdash e :: c}{\Gamma \vdash e\ \textbf{where}\ d :: c}\ \text{C-Rhs}$$

$$\frac{\Gamma \vdash e :: c}{\Gamma \vdash \mid g = e\ \textbf{where}\ d :: c}\ \text{C-GRhs}$$

Figure 4.9: Contracting rules for right-hand sides.

Right-hand sides. Right-hand side contracting rules (see Figure 4.9) are equally simple: the contract of e is looked up in the contract environment and is used for the entire right-hand side. The same is done for guarded expressions (Figure 4.10).

$$\frac{\Gamma \vdash e :: c}{\Gamma \vdash g \mid e :: c}\ \text{C-GExpr}$$

$$\frac{\Gamma \vdash g :: c}{\Gamma \vdash (g : gs) :: c}\ \text{C-GCons}$$

$$\frac{fresh(i)}{\Gamma \vdash [] :: true_i}\ \text{C-GNil}$$

Figure 4.10: Contracting rules for guarded expressions.

Guarded expressions. Figure 4.10 illustrates the contracting rules for guarded expressions. The guard itself is ignored, and the contract for the expression that is guarded is used for the entire guarded expressions.

Expressions. Figure 4.11 details the contracting rules for expressions, which have large overlaps with those for patterns. Note how there is no rule `C-InfixApp`. This is because infix function applications are converted to regular function applications before contract inference occurs. As such, there is no need for such a rule.

Rules that are either trivial or similar to those of patterns are: `C-Hole`, `C-Feedback`, `C-MustUse`, `C-Con`, `C-Lit`, `C-App`, `C-Paren`, `C-Tuple`, `C-Cons`, `C-Nil` and `C-Neg`. The rest of the contracting rules are discussed hereafter.

`C-Case` : This rule describes contracting of a case statement for an arbitrary number of cases. The contract of the expression being evaluated, m, must be the same as the contracts of the patterns p_i, namely c_1. The contract of the entire case statement is c_2, which is the contract that is given to each alternative.

`C-If` : Syntactic sugar for a boolean case statement. Like its generalized version, the contract of the entire statement is the same of each alternative, in this case c.

`C-Lambda` : This contracting rule is the same as in Stutterheim's work.

`C-Let` : A non-generalized let. Generalization is not necessary because every contract variable is implicitly universally quantified in the global contract environment.

`C-Enum` : A list generator. y and z are fully optional, but if they are present, they must have the same contract as that of x, namely c. A fresh outer contract is generated to complete the final contract.

$$\frac{fresh(i)}{\Gamma \vdash ? :: true_i}\ \text{C-Hole}$$

$$\frac{\Gamma \vdash m :: c_1 \qquad \forall i \in [0 \ldots n] \Gamma \vdash p_i :: c_1 \qquad \forall i \in [0 \ldots n] \Gamma \vdash e_i :: c_2}{\Gamma \vdash \textbf{case}\ m\ \textbf{of}\ \{p_0 \rightarrow e_0; ...; p_n \rightarrow e_n\} :: c_2}\ \text{C-Case}$$

$$\frac{\Xi_{ctrt}(x) = c}{\Gamma \vdash x :: c}\ \text{C-Con}$$

$$\frac{\Gamma \vdash t :: c \qquad \Gamma \vdash e :: c}{\Gamma \vdash \textbf{if}\ cond\ \textbf{then}\ t\ \textbf{else}\ e :: c}\ \text{C-If}$$

$$\frac{\Gamma[x \mapsto c_1] \vdash e :: c_2}{\Gamma \vdash \lambda x \rightarrow e :: c_1 \rightarrowtail c_2}\ \text{C-Lambda}$$

$$\frac{\Gamma \vdash e_2 :: c}{\Gamma \vdash \textbf{let}\ \mathbf{x} = e_1\ \textbf{in}\ e_2 :: c}\ \text{C-Let}$$

$$\frac{l \text{ is a literal} \qquad fresh(i)}{\Gamma \vdash l :: literal_i}\ \text{C-Lit}$$

$$\frac{\Gamma \vdash e_1 :: c_2 \rightarrowtail c \qquad \Gamma \vdash e_2 :: c_2}{\Gamma \vdash e_1\ e_2 :: c}\ \text{C-App}$$

$$\frac{\forall i \in [0 \ldots n] \Gamma \vdash e_i :: c_i \qquad fresh(j)}{\Gamma \vdash (e_0, .., e_i) :: true_j \ \langle @@ \rangle\ (c_1, .., c_i)}\ \text{C-Tuple}$$

$$\frac{\Gamma(x) = c}{\Gamma \vdash x :: c}\ \text{C-Var}$$

$$\frac{fresh(i)\ \Gamma \vdash x :: c \qquad \Gamma \vdash y :: c \qquad \Gamma \vdash z :: c}{\Gamma \vdash [x\ ..\ (y)\ ..\ (z)] :: true_i \ \langle @ \rangle\ c}\ \text{C-Enum}$$

$$\frac{\Gamma \vdash x :: c \qquad \Gamma \vdash xs :: list \ \langle @ \rangle\ c}{\Gamma \vdash (x : xs) :: list \ \langle @ \rangle\ c}\ \text{C-Cons}$$

$$\frac{fresh(i, j)}{\Gamma \vdash [] :: true_i \ \langle @ \rangle\ true_j}\ \text{C-Nil}$$

Figure 4.11: Contracting rules for expressions.

$$\frac{fresh(i)}{\Gamma \vdash ? :: true_i}\ \text{C-AHole}$$

$$\frac{fresh(i)}{\Gamma \vdash :: true_i}\ \text{C-AltEmpty}$$

$$\frac{\Gamma \vdash rhs :: c}{\Gamma \vdash p \rightarrow rhs :: c}\ \text{C-Alt}$$

$$\frac{\Gamma \vdash a :: c \qquad \Gamma \vdash as :: c}{\Gamma \vdash (a : as) :: c}\ \text{C-ACons}$$

$$\frac{fresh(i)}{\Gamma \vdash [] :: true_i}\ \text{C-ANil}$$

Figure 4.12: Contracting rules for alternatives.

Alternatives. The contracting rules for alternatives in Figure 4.12 are quite simple. `C-AHole` and `C-AltEmpty` are trivial, and `C-Alt` takes the contract of the right-hand side as the contract of the entire alternative. `ACons` and `ANil` are how the `Alts` node functions behind the scenes: all alternatives must have the same contract.

Substitutions and unification

We use the same data type for substitutions as Stutterheim. Our unification algorithm, too, is identical to that of Stutterheim, which itself is based upon Robinson's unification algorithm. We refer you to subsection 3.4.2 of Stutterheim's thesis for a description of the unification algorithm and the substitutions it generates. When you are done reading, we will note a caveat in his description of the algorithm.

It merits mentioning that although Stutterheim's unification algorithm supports concrete contract refinement in theory, in code this is unsupported. In fact, the example posed by Stutterheim:

$\mathcal{U}\ (int, nat)$

results in an error message, as evidenced by the relevant pattern matches of the `unifyC` function as shown in Figure 4.13.

This is because the *int* and *nat* contracts are in fact defined as user-defined concrete contracts. The default contracts that Stutterheim refers to in Figure 3.2 of his thesis do not exist in the actual Contract data type in the source code. It only supports the inhabitants of our simplified grammar as shown in Figure

```
unifyC :: Contract → Contract → Subst
unifyC c1              c2 | c1 == c2   = SId
unifyC v@CVar{}        c               = unifyCVars v c
unifyC c               v@CVar{}        = unifyCVars v c
...
unifyC c1 c2 = unifyErr "No such unification case when" c1 c2
```

Figure 4.13: A part of the `unifyC` function of Algorithm $\mathcal{CW}$.

4.3. Relations between contracts that allow for refinement are thus non-existent. While these relations can be mapped out manually, a better idea would be to look towards true refinement types as described in section 2.3.

The manner in which we apply generated substitutions differs greatly from Stutterheim's approach. We explain how and why in section 4.7.

Algorithm $\mathcal{CHW}$

Now that we have reviewed the contracting rules and the unification algorithm, let us define the contract inference Algorithm $\mathcal{CHW}$, which is based upon Stutterheim's Algorithm $\mathcal{CW}$. Our algorithm takes a `Module` and infers contracts for each declaration, pattern, right-hand side, (guarded) expression and alternative that resides in the module.

To infer a contract, we require:

1. The contract environment Γ.
2. The type source Ξ.
3. An infinite supply of fresh *true* contracts.

We follow the same order as the contracting rules here: patterns are introduced first, then declarations, function bindings, right-hand sides, expressions and alternatives.

Patterns. The code for patterns is shown in Figure 4.14. We omit the infix constructor, as it is the same as in Stutterheim's system for expressions. Again, `PList` is replaced by the `PCons` and `PNil` constructors, which is how the constructor works behind the scenes. Note how patterns do not generate any substitutions, as no unifications are required: all contracts come from the type source Ξ, which uses the type of a pattern to generate an initial contract. These

types come from the type inference algorithm, where they have already been unified.

```
chw :: Environment → TypeSource → Pattern → (Substitution, Contract)

chw Γ Ξ PHole =
 let i be fresh
 in (Id, true_i)

chw Γ Ξ PWildcard =
 let i be fresh
 in (Id, true_i)

chw Γ Ξ (PCon n p) = (Id, Ξ(PCon n p))

chw Γ Ξ PNil =
 let i and j be fresh
 in (Id, true_i <@> true_j)

chw Γ Ξ (PCons x xs) =
 let (θ1,c) = chw Γ Ξ x
     (θ2, true_i <@> c) = chw Γ Ξ xs
 in  (θ2 ∘ θ1, true_i <@> c)

chw Γ Ξ (PLit l) =
 let i be fresh
 in (Id, literal_i)

chw Γ Ξ (Paren p) = chw Γ Ξ p

chw Γ Ξ (Tuple (p:ps)) =
 let i be fresh
     (θ1, c1) = chw Γ Ξ p
     (θ2, c2) = chw Γ Ξ ps
 in  (θ2 ∘ θ1, true_i <@@> (c1, c2))

chw Γ Ξ (Tuple [p]) = chw Γ Ξ p

chw Γ Ξ (PVar x) = (Id, Γ(x))

chw Γ Ξ (PAs p ps) = (Id, Γ(p))
```

Figure 4.14: Algorithm $\mathcal{CHW}$ for patterns.

Declarations. For declarations (Figure 4.15), we omit `DHole` and `DEmpty`, as they both function the same as `PHole`. `DCons` is of particular interest, as it shows how substitutions are passed between groups of function bindings.

Function bindings. Figure 4.16 shows Algorithm $\mathcal{CHW}$ for function bindings. We omit `FBHole` as we did before, and `FCons` and `FNil` are trivial. `FunBind` may look a bit odd. This is what it does: first, the contract of the function identifier, c_1, is trimmed by the length of *pats* starting from the left of the contract, removing these arguments from the contract and producing c_1', the contract for the result of the function. Then, c_1' is unified with c_2, which is the contract for the right-hand side of the function binding.

```
chw :: Environment → TypeSource → Declaration → (Substitution, Contract)

chw Γ Ξ (DFunBinds []) = chw Γ Ξ DEmpty

chw Γ Ξ (DFunBinds fs) = chw Γ Ξ fs

chw Γ Ξ (DPatBind p rhs) =
 let (θ1, c1) = chw Γ Ξ p
     (θ2, c2) = chw Γ Ξ rhs
     θ3 = 𝒰(c1, c2)
 in  (θ3 ∘ θ2 ∘ θ1, θ3 ∘ θ2 c1)

chw Γ Ξ (DCons d ds) =
 let (θ1, c1) = chw Γ Ξ d
     (θ2, c2) = chw (θ1 Γ) Ξ ds
 in  (θ2 ∘ θ1, c1)

chw Γ Ξ DNil =
 let i be fresh
 in  (Id, true_i)
```

Figure 4.15: Algorithm $\mathcal{CHW}$ for declarations.

```
chw :: Environment → TypeSource → FuncBinding → (Substitution, Contract)

chw Γ Ξ (FunBind nm pats rhs) =
 let c1 = Γ(nm)
     lp = length pats
     c1' = dropc lp c1
     (θ1, c2) = chw Γ Ξ rhs
     θ2 = 𝒰(c1', c2)
 in  (θ2 ∘ θ1, θ2 c1)

chw Γ Ξ (FCons f fs) =
 let (θ1, c1) = chw Γ Ξ f
     (θ2, c2) = chw Γ Ξ fs
 in  (θ2 ∘ θ1, c1)

chw Γ Ξ FNil =
 let i be fresh
 in  (Id, true_i)
```

Figure 4.16: Algorithm $\mathcal{CHW}$ for function bindings.

Right-hand sides. The code for right-hand sides (figue 4.17) is quite simple. In both cases, the contract of expression e is used for the entire right-hand side. In the case of guarded expressions, the substitution of each guarded expression is collected.

```
chw :: Environment → TypeSource → RHS → (Substitution, Contract)

chw Γ Ξ (Rhs e w) =
 let (θ1, c1) = chw Γ Ξ e
     (θ2, c2) = chw Γ Ξ w
 in  (θ2 ∘ θ1, c1)

chw Γ Ξ (GRhs (e:es) w) =
 let (θ1, c1) = chw Γ Ξ e
     (θ2, c2) = chw Γ Ξ es
     (θ3, c3) = chw Γ Ξ w
 in  (θ3 ∘ θ2 ∘ θ1, c1)

chw Γ Ξ (GRhs [] w) =
 let i be fresh
 in  (Id, true_i)
```

Figure 4.17: Algorithm $\mathcal{CHW}$ for right-hand sides.

Expressions. We omit the code for `Hole`, `Feedback`, `MustUse`, `Neg` and `Paren` in Figure 4.18, as these are trivial: in the first case, a fresh *true* contract is generated and in the other cases, the expression e that is wrapped by the node is used to attain the substitutions and contract. Furthermore, we omit `InfixApp` because it is converted to regular function application, which is covered by `App`.

The code for `App` differs from the original algorithm. Instead of generating a fresh *true* contract to unify with the contract of f, we drop a single argument from the c_1 contract to obtain α, the contract for the result of f. This is done to reduce the amount of generated substitutions. α is then used for unification in the usual manner.

Alternatives. For alternatives, we omit `AHole` and `AltEmpty` in Figure 4.19 because they are trivial.

4.6 Code Generation

After the process of contract inference, we possess enough information to fully contract the original code. For each function definition `X`, we generate three new function definitions:

```
chw :: Environment → TypeSource → Expression → (Substitution,Contract)

chw Γ Ξ (Case e alts) =
 let (θ1,c1) = chw Γ Ξ e
     (θ2,c2) = chw Γ Ξ alts
 in  (θ2 ∘ θ1, θ2 ∘ θ1 c2)

chw Γ Ξ (Con name) = (Id, Ξ(name))

chw Γ Ξ (Lit l) = (Id, Ξ(l))

chw Γ Ξ (If i t e) =
 let (θ1,c1) = chw Γ Ξ i
     (θ2,c2) = chw Γ Ξ t
     (θ3,c3) = chw Γ Ξ e
     θ4 = 𝒰(c2,c3)
 in  (θ4 ∘ θ3 ∘ θ2 ∘ θ1, θ4 ∘ θ3 ∘ θ2 ∘ θ1 c2)

chw Γ Ξ (Lambda p e) =
 let (θ1,c1) = chw Γ Ξ p
     (θ2,c2) = chw Γ Ξ e
 in  (θ2 ∘ θ1, θ2 (c1 ⤚ c2))

chw Γ Ξ (Let decls e) =
 let (θ1,c1) = chw Γ Ξ decls
     (θ2,c2) = chw Γ Ξ e
 in  (θ2 ∘ θ1, c2)

chw Γ Ξ (App f x) =
 let (θ1,c1) = chw Γ Ξ f
     (θ2,c2) = chw Γ Ξ x
     α = dropc 1 c1
     θ3 = 𝒰(θ2 c1,c2 ⤚ α)
 in  (θ3 ∘ θ2 ∘ θ1, θ3 α)

chw Γ Ξ (Enum f t o) =
 let (θ1,c1) = chw Γ Ξ f
     (θ2,c2) = chw Γ Ξ t
     (θ3,c3) = chw Γ Ξ o
     θ4 = 𝒰(c1,c2)
     θ5 = 𝒰(θ4 c2,c3)
 in  (θ5 ∘ θ4 ∘ θ3 ∘ θ2 ∘ θ1, θ5 ∘ θ4 ∘ θ3 ∘ θ2 ∘ θ1 c1)

chw Γ Ξ (Cons x xs) =
 let i be fresh
     (θ1,c1) = chw Γ Ξ x
     (θ2,c2) = chw Γ Ξ xs
     θ3 = 𝒰(c2,θ2 (true_i <@> c1))
 in  (θ3 ∘ θ2 ∘ θ1, θ3 (true_i <@> c1))

chw Γ Ξ Nil =
 let i and j be fresh
 in  (Id, true_i <@> true_j)

chw Γ Ξ (Tuple (t:ts)) =
 let i be fresh
     (θ1,c1) = chw Γ Ξ t
     (θ2,c2) = chw Γ Ξ ts
 in  (θ2 ∘ θ1, true_i <@@> (c1,c2))

chw Γ Ξ (Tuple [t]) = chw Γ Ξ t
```

Figure 4.18: Algorithm $\mathcal{CHW}$ for expressions.

```
chw :: Environment → TypeSource → Alternative → (Substitution, Contract)

chw Γ Ξ (Alt p rhs) =
 let (θ1, c1) = chw Γ Ξ p
     (θ2, c2) = chw Γ Ξ rhs
 in  (θ2 ∘ θ1, θ2 c2)

chw Γ Ξ (ACons x xs) =
 let (θ1, c1) = chw Γ Ξ x
     (θ2, c2) = chw Γ Ξ xs
     θ3 = 𝒰(c1, c2)
 in  (θ3 ∘ θ2 ∘ θ1, θ3 ∘ θ2 c1)

chw Γ Ξ ANil =
 let i be fresh
 in  (Id, true_i)
```

Figure 4.19: Algorithm $\mathcal{CHW}$ for alternatives.

- ___*final*_$\boxed{\text{X}}$
- ___*app*_$\boxed{\text{X}}$
- ___*contracted*_$\boxed{\text{X}}$

We will inspect each template in detail and explain its purpose.

4.6.1 ___final_$\boxed{\text{X}}$

The ___*final*_$\boxed{\text{X}}$ template is simple: it is a transformed copy of the original function definition. Every function application found within the original code is replaced by its contracted equivalent, the ___*app*_$\boxed{\text{X}}$ template. This transformation makes this template the entry point for all other generated functions.

As an example, here is the filled-in ___*final*_$\boxed{\text{X}}$ template for the $\boxed{\texttt{foldr}}$ function:

```
__final_foldr __ctrtf f b (x:xs) = f x (__app_foldr ctrt pos (p1, __ctrtf) (
    p2, b) (p3, xs))
__final_foldr __ctrtf f b [] = b
```

The first thing to note is how the recursive call to $\boxed{\texttt{foldr}}$ is contracted by replacing the original function application with a call to the ___*app*_$\boxed{\text{X}}$ template of the $\boxed{\texttt{foldr}}$ function definition. From the original code, extra information is generated that is passed to the ___*app*_$\boxed{\text{X}}$ template. The placeholders in the above example represent the following extra information:

- *ctrt*: The contract that will be used to assert the original function. Details of the generation of the contract are found in the next section (4.7).

- *pos*: A tuple of the line and column position of the function application. It is is used to generate specific feedback at runtime in case of a contract violation.
- $p1...n$: Line and column position information is generated for each argument applied to the original function application, again used for specific feedback.

Patterns representing functions are already contracted

It may puzzle you why the application of `f` at the beginning of the right-hand side is not transformed in the same manner. This is because it is a *pattern* available in the left-hand side of the function definition. It is a call to a locally-available function that has already been contracted.

To clarify, let us look at this example. The following expression:

```
foldr insert [] [5,4,7,0,10]
```

is transformed into the following one:

```
__app_foldr (p1,__contracted_insert ctrt pos) (p2,[]) (p3,[5,4,7,0,10])
```

Because `insert` is a function that does not yet have any arguments applied to it, it is replaced by its ___*contracted*_`X` counterpart.

As you can see, the ___*contracted*_`X` template of `insert` is passed to the ___*app*_`X` template of `foldr`. There, the contracted version of `insert` is annotated with detailed information and wrapped so that it can be applied with regular function application instead of using the `appParam` function from the *typed-contracts* library. Hence, the *pattern* `f` in `___final_foldr` in our previous example does not have to be transformed again.

Type difference between contracted and regular functions

In `___final_foldr`, `f` has a regular function type (`a → b → b`), but `___app_foldr` expects a contracted version of `f` with the type `a ⤜ b ⤜ b`. So, the ___*final*_`X` template includes a new pattern for each function argument present in the original function definition. In the case of `foldr`, this is only `f`. The original pattern is prefixed with `___ctrt` and added to the function definition.

4.6.2 ___app_X

This template is given information for use in feedback, wraps it appropriately using `appParam`, and passes it to the ___*contracted*_X template of the function definition.

Let us examine the template:

```
__app_X ctrt posinfo patterns = righthandside
```

The placeholders *patterns* and *righthandside* are relatively simple:

- *patterns* generates a tuple for each argument, containing the position information of the argument and the argument itself.
- *righthandside* uses this extra information to provide richer feedback at runtime and applies the arguments to the ___*contracted*_X template of the function.

Let us take the following example:

```
g f x = f x
```

which generates the following code:

```
__app_g ctrt posinfo (posf,f) (posx,x) =
 appParam (appParam (__contracted_g ctrt posinfo) (show f ++
     generatePositionData posf) f) (show x ++ generatePositionData posx)
     x
```

The ___*contracted*_X template of the function is fed arguments using `appParam`, and are accompanied by feedback strings containing the argument as a string and its position in the source code.

4.6.3 ___contracted_X

The final template, ___*contracted*_X, constructs an assertion with the contract provided for a function definition X.

Let us inspect the template:

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

The placeholders represent the following:

- *function info*: A string that informs the user which function violated its contract, and if that function is higher-order or not.
- *contracted function definition*: Using the `fun` function, a function is lifted to a contracted version.

The latter placeholder benefits from a few examples. Here is what the contract version of the identity function looks like:

```
fun (λx → __final_id x)
```

For each argument, an extra layer of the `fun` function is applied to capture them and make them available to the original function.

Functions as arguments to higher-order functions

A function argument is slightly more involved, as it must be applied to its arguments using `appParam`. For example, the code snippet

```
g f x = f x
```

generates the following for the *contracted function definition* placeholder:

```
(fun (λf → (fun (λx → __final_g f (λa → (appParam f info a)) x))))
```

Note how `a` is captured and applied to `f` using `appParam`. This is because `f` is a contracted function that was passed to the higher-order function.

Two versions of function arguments

Because `a` is captured using a lambda function, the type of the expression `(λa → (appParam f info a))` is `a → a`, and can be used as a "regular" function in `___final_g`.

`f`, whose type is `a ⤚ a`, is also provided to `___final_g` unwrapped, in the case that a higher-order function in the right-hand side of `___final_g` takes `f` as an argument. A good example of this is that of `foldr`:

```
__final_foldr __ctrtf f b (x:xs) = f x (__app_foldr ctrt pos (p1,__ctrtf) (
    p2,b) (p3,xs))
__final_foldr __ctrtf f b [] = b
```

4.7 Generation of final contracts

In this section, we illustrate an important limitation in the substitutions generated by Algorithm $\mathcal{CW}$ and propose a solution to this limitation. This limitation manifests itself when generating code for a library. We have seen how templates for the *typed-contracts* library are filled out, but it is not yet clear how the intermediate contract language is translated to a *typed-contracts* contract. Hence, that is what we start with before moving on to the limitation.

4.7.1 Translation of intermediate contract language to *typed-contracts* library

Because the intermediate contract language is inspired by the notation used by the *typed-contracts* library, translation between the two is simple.

The $true_i$ and $false_i$ contracts are translated into the `true` and `false` functions, respectively. The index i is only relevant to contract inference, so it is discarded.

The contract arrow ($\rightarrowtail$), functor ($<@>$) and bifunctor ($<@@>$) translate directly to the *typed-contracts* library.

At the moment, $literal_i$ contracts are translated into `true` contracts. Stutterheim's thesis suggests constant expression contracts as a better candidate for translation, but notes that extra program analysis is required to safely replace a simple `true` contract with a constant expression one.

Finally, user-defined concrete contracts only exist as a string in the intermediate contract language. Translation is done simply by printing that string when generating the code, as the string is the identifier of a contract defined outside of the generated code.

Now that it is clear how translation works, we explain the limitation in Algorithm $\mathcal{CW}$ and how doing the translation we have just discussed at runtime proves troublesome.

4.7.2 The monomorphic Algorithm $\mathcal{CW}$

Algorithm $\mathcal{CW}$ contains a severe limitation: every generated substitution is placed in a global set. This set is then used to generate the final contract of an expression e during code generation by applying the substitutions to the

inferred contract of e. The limitation presents itself when we apply a function `f` to two different arguments with different contracts:

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

Listing 4.5: An example code snippet that may generate incorrect code when using substitutions from Algorithm $\mathcal{CW}$.

If we leave the contract environment Γ empty, both `'a'` and `5` receive a *true* contract, say, $true_1$ and $true_2$. During inference, these contracts are unified and generate a substitution indicating they refer to the same contract. In the end, the contract of `z` is as follows:

```
true₀ <@@> (true₃ <@> true₁, true₄ <@> true₂)
```

Because *true* contracts are translated into the `true` *typed-contracts* contract, no runtime or compilation error occurs.

But if we seed Γ with an initial contract, the story is entirely different! The contract of `z` is now set to the following:

```
Γ(z) = true₀ <@@> (true₁ <@> isChar, true₂ <@> isNum)
```

Listing 4.6: Contract environment Γ is now seeded with a contract for `z`.

For the rest of the identifiers and literals, the following contracts are constructed by Algorithm $\mathcal{CW}$:

```
Γ(x)    = true₄
Γ(f)    = (true₄ ↣ (true₃ <@> true₄))
Γ('a')  = true₅
Γ(5)    = true₆
```

The relevant unifications of `f 'a'`, `f 5` and `z` are as follows:

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

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

We obtain the following substitutions:

- $\boxed{\texttt{true}_4 \mapsto \texttt{true}_5}$
- $\boxed{\texttt{true}_4 \mapsto \texttt{true}_6}$

We can apply these substitutions in left-to-right or right-to-left order to the contract of the right-hand side of z, which is

`(true`$_7$ `<@@> true`$_3$ `<@> true`$_4$`, true`$_3$ `<@> true`$_4$`)`. *true*$_7$ is a fresh contract:

- left-to-right: `(true`$_7$ `<@@> (true`$_3$ `<@> true`$_5$`, true`$_3$ `<@> true`$_5$`))`
- right-to-left: `(true`$_7$ `<@@> (true`$_3$ `<@> true`$_6$`, true`$_3$ `<@> true`$_6$`))`

What happens when we unify either of these with the contract for `z`?

```
unify (true0 <@@> (true1 <@> isChar, true2 <@> isNum))
      (true7 <@@> (true3 <@> true5, true3 <@> true5))
```

Relevant substitutions:

- `true`$_5$ $\mapsto$ `isChar`
- `true`$_5$ $\mapsto$ `isNum`

```
unify (true0 <@@> (true1 <@> isChar, true2 <@> isNum))
      (true7 <@@> (true3 <@> true6, true3 <@> true6))
```

Relevant substitutions:

- `true`$_6$ $\mapsto$ `isChar`
- `true`$_6$ $\mapsto$ `isNum`

Here, the limitation becomes apparent: regardless of how we apply these substitutions, one of the contracts of expressions `f 'a'` and `f 5` will always be incorrect! If we apply the substitutions left-to-right, the expression `f 5` is given the contract `true`$_1$ `<@> isChar`. Doing it right-to-left provides `f 'a'` with the contract `true`$_1$ `<@> isNum`.

In both instances, either a compile-time or a runtime error occurs. This breaks Stutterheim's proposition that *"(an) inferred contract will never fail assertion for that expression"* ([17], proposition 6).

In this manner, Algorithm CW is monomorphic: the inferred contract of a function may constrain the acceptable values of an argument to a static subset of what the argument should actually be able to accept.

Proposed solution: bottom-up and top-down substitution lists

Instead of a single, global set of substitutions that is applied to every contract during translation, we keep track of a single bottom-up list of substitutions and several top-down lists. Contract variables are also given a flavour of top-down or bottom-up: $true_i$ contracts are bottom-up, concrete contracts are top-down.

Whenever unification between two contracts takes place, the resulting substitutions are split up in a bottom-up and top-down group:

- The bottom-up group holds all substitutions that go from bottom-up contract variables to other bottom-up contract variables.
- The top-down group holds all substitutions that go from bottom-up contract variables to top-down contract variables, as well as those that go from top-down contract variables to other top-down contract variables.

The bottom-up group is added to the bottom-up substitution list. The top-down group is added to the top-down substitution list, which already contains the parent's top-down substitution list. In other words, the bottom-up group travels up the AST towards the top, and the top-down group travels downwards into the child nodes of the node where unification took place.

Applying bottom-up and top-down substitution lists

The algorithm for calculating the final contract from these lists is comprised of four steps. First, look up the initial contract $c_{initial}$ in the contract environment Γ.

Second, apply the bottom-up substitution list $s_{bottom-up}$ to $c_{initial}$ until a fixed point is reached or until $n + 1$ applications have taken place, where n is the number of substitutions in the bottom-up list. This yields $c_{bottom-up}$. For those interested, a more detailed description of this application can be found in section 5.5. The reason why the substitutions must be applied up to $n + 1$ times is because, in our language, the ordering of function declarations is random, but the order in which the generated substitutions must be applied is not. Stutterheim's language uses nested let-expressions to introduce multiple function declarations, which means the order of the generated substitutions is always correct, so it did not have to perform this search.

As the third step, the top-down substitution list $s_{top-down}$ is updated to reflect the changes in $c_{initial}$ that have been introduced by $s_{bottom-up}$. This is done by

applying $s_{bottom-up}$ to $s_{top-down}$ up to $n+1$ times until a fixed point is reached, producing $s_{updated}$.

Finally, $s_{updated}$ is applied to $c_{bottom-up}$ in the same manner until a fixed point is reached, resulting in c_{final}.

At first, it appears logical to apply these substitutions during code generation, and not during runtime, but this is not optimal. To see why, let us look at the code generated by this example:

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

Listing 4.7: Expanded version of listing 4.5.

The example has been expanded to include function `g`, which is called by `f`. Our code generation system for the *typed-contracts* library produces the following ___final_`X` code for `z`, `g` and `f` (we omit the line and column position information):

```
__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)
```

Listing 4.8: Output of code generation for the *typed-contracts* library.

Calling the contracted version of `f` will call the contracted version of `g`, but with an initial contract! In other words, the top-down substitutions that made the contract passed to `__app_f` more specific, are not utilized by `__app_g`. The solution appears trivial: pass the top-down substitutions as another argument to the ___app_`X` and ___contracted_`X` templates and apply them to the initial contract at runtime.

Applying substitutions at runtime

Unfortunately, it is not so simple. Doing substitutions at runtime instead of statically means a runtime translation is necessary between the intermediate contract language and the *typed-contracts* contract datatype, because the substitutions only work on the intermediate contract language. The targeted library expects a contract in its own datatype, of course. The intermediate contract language is modeled as a simple data type automatically generated by the Utrecht University Attribute Grammar Compiler (UUAGC) without a phantom type, and *typed-contracts*'s datatype is a GADT with a phantom type used for a type-level representation of the contract. Conversion between the two means we need

to be able to tell what this phantom type should be. This is especially a problem when we need to convert between user-defined concrete contracts (`Prop` and `PropInfo`), because each of these contracts may have a different phantom type and assorted type class constraints, and the only information available in the AST datatype equivalent is a `String`.

Several attempts were made to provide a runtime translation function, but none were successful. Due to time constraints, we defer this translation method as future work. Section 5.4 provides an overview of the attempts and promising methods.

The current code generation system generates monomorphic code that may fail when a contracted function is applied to different contracts.

4.8 Providing richer feedback

We investigate how feedback towards the user can be improved in this section.

4.8.1 Modifications to *typed-contracts* library

In order to provide adequate feedback to the user, we modify the *typed-contracts* library by Hinze et al. [10] to produce more specific feedback in a language that the user can understand. The original library code provides error messages that refer to contract violations and (higher-order) blame of expressions, concepts that are foreign to the average user and thus do not provide any useful feedback.

Here is an example of the feedback provided by the original library:

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

> app inc 1 (−5)
> *** Exception: contract failed: the expression labelled ‘1’ is to blame.
```

Type position information

To improve this rudimentary feedback, we require position information that records the position of an argument (or result) in a type. We record this information in a tuple:

```
type PosInfo = (Int,Int)
```

For example, the position $(0, 1)$ indicates the first argument of a function that has one argument. The position $(1, 1)$ indicates the result of such a function.

Using this information, it is possible to produce highly specific feedback messages:

```
posInfoText (pos,arity) | pos < arity  = "The " ++ showPos (pos+1) ++ "
    supplied argument of this function "
                        | pos == arity = "The result of this function "
                        | (pos,arity) == (-1,-1) = "An unknown position "

showPos p | p == 1 = "first"
          | p == 2 = "second"
          | p == 3 = "third"
          | p == 4 = "fourth"
          | p == 5 = "fifth"
          | p == 6 = "sixth"
          | otherwise = show p
```

Listing 4.9: Code that generates text indicating the location of the problem.

Using the type position information

The `Contract` GADT is modified to include the `PropInfo` constructor, which takes an additional argument: a function that, given type position information of which argument or result that the property is asserting, returns an appropriate error message:

```
PropInfo   ::  (aT → Bool) → (PosInfo → String) → Contract aT
Prop       ::  (aT → Bool) → Contract aT
```

Here is a usage example of `PropInfo`:

```
isBiggerThan_prop = PropInfo (λx → fromEnum x > 5 ) (λp → mkErrorMsg p "
    the number must be larger than five.")
```

If this property were to be asserted on the first argument of a function and was violated, it would generate the following error message:

> *"The first supplied argument of this function does not fulfill the following property: the number must be larger than five."*

We add the functions `assertPos` and `assert''` to the *typed-contracts* library to record this type position information. For brevity, we omit the definitions of `makeLoc` and `Def`. These can be found in the *typed-contracts* library source code.

```
assertPos :: String → String → Contract a → a → a
assertPos s ext c = assert" c (0, ctrtarity c) (makeloc (Def (s ++ ext)))
 where
  ctrtarity :: Contract a → Int
  ctrtarity (Function c1 c2) = 1 + ctrtarity (c2 undefined)
  ctrtarity _                = 0
```

External feedback

`assertPos`'s purpose is twofold: to initialize the type position information, and to accept an additional string `ext`, which is used during feedback.

Because the `app` function is responsible for providing a location label in the form of an integer, it makes sense to define a version of `app` that accepts an additional feedback string instead of an integer: `appParam`.

```
app :: (a ↣ b) → Int → a → b
app f loc x = apply f (makeloc (App loc)) x

appParam :: (a ↣ b) → String → a → b
appParam f s x = apply f (makeloc (Def s)) x
```

The use for these extra feedback strings is discussed in the next subsection.

4.8.2 External library-agnostic feedback

Along with improvements to the *typed-contracts* library, we have identified several techniques for generating library-agnostic feedback through abstract syntax tree analysis and modification.

Dynamic representation of values using `Show` class

The most robust way of displaying a value is through use of the `Show` typeclass. The natural caveat one thinks of is the lack of a `Show` instance for terms such as functions. Circular or infinite data structures are also troublesome.

So, when generating code that relies on `Show` instances, one must ensure that there are no datatypes without a `Show` instance, and that circular or infinite data structures are not present.

In our particular situation, we assume that every argument has a `Show` instance. Because QuickCheck should not generate infinite structures (quite the contrary!), this appears to be a reasonable assumption.

`Show` instances for functions would still be required, but those can be provided:

```
instance Show (a ⤖ b) where
        showsPrec a = showString "<function>"
```

Static string representation of functions

In the event of a contract violation inside a higher-order function, it is possible that the offending value is a function. But providing a meaningful `Show` instance for the function type is impossible. How can this limitation be amended? One possibility is to generate a static string representation of each argument and use that in the feedback. For instance, the following example:

```
f g x = g x
z = f id 1
```

would become

```
f g x = g (x,"x")
z = f (id,"id") (1,"1")
```

However, a string representation loses all notion of the original structure. In the case of functions, this is not a problem, but structures such as lists suffer. Let us assume we apply the `map` function to an appropriate function and the list `[1,2,3]`. Code is generated that replaces the original argument with the tuple `([1,2,3],"[1,2,3]")`. The `map` function then applies its function argument to the first item in the list. This list item causes a contract violation, but the only visual information available is a string representation of the entire list. Clearly, this catch-all approach does not work.

By defining a new typeclass `ShowFunc`, we can circumvent this issue while still making use of the statically generated string representations:

```
class ShowFunc a where
 showFunc :: String → a → String

instance (Show a) ⇒ ShowFunc a where
 showFunc _ a = show a

instance ShowFunc (a → b) where
 showFunc rep _ = rep
```

We leave integrating this technique as future work.

Identifying a function as higher-order

By inspecting the type of a function, it is possible to inform the user if the contract violation was caused by a higher-order function or a first-order function.

Dynamic generation of line and column numbers of an expression

If the abstract syntax tree provides them, extracting line and column numbers and putting them in the modified AST as a string is trivial and a highly valuable way of improving feedback.

4.8.3 Example of detailed feedback for *typed-contracts* library

While we are unable to say what function caused the contract violation in a higher-order function due to the lack of user-defined typeclasses, we can indicate which arguments of both the higher-order function and the function argument failed. We use this information to generate highly detailed feedback, as can be seen with this snippet of generated code:

```
g f x = f x
```

becomes

```
(fun (λf → (fun (λx → g (λa → (appParam f (concat "the application of
    the higher-order function 'g' ",(generatePositionData posinfo),". g
    has a function as its first argument."," The first argument of that
    function",", namely ",show a]) a)) x))))
```

Listing 4.10: Detailed feedback is generated when applying the function argument in a higher-order function.

Note how the argument `x` is applied to `f` with a very detailed feedback message that incorporates the following elements:

- Whether the function is higher-order or not.
- Run-time generation of line and column number information.
- Identification of the position of `f` in the type of `g`. This is a duplication of the type position information functionality added to the *typed-contracts* library.
- Identification of the position of `x` in the type of `f`.

- Run-time representation of the offending value using the `Show` typeclass.

In the case of a contract violation, the feedback could look like this:

> *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 higher-order function 'g' at line number 1, column number 1. g has a function as its first argument. The first argument of that function, namely -5, does not fulfill the following property: the number must be a natural number.*

From this feedback, the user can deduce the following:

- Which application of a higher-order function caused the contract violation.
- Which argument of the higher-order function is the function that caused the contract violation.
- Which argument of that function caused the violation.
- Possibly, the offending value.

4.8.4 Reusing QuickCheck properties

The Ask-Elle programming tutor supports QuickCheck properties that are executed to verify the correctness of code that cannot be reduced to a known strategy. A very interesting idea is to derive contracts from such properties automatically and use them to construct contracts that can be used to contract the result of the function. However, remember that we are limited to non-dependent contracts, so we only have access to a single argument or the result of the function in a contract. Most QuickCheck properties, however, refer to one or more arguments of the function.

A simple QuickCheck property of the palindrome function illustrates the problem well:

```
prop_Main = λxs → whenFail (putStrLn "This function does not correctly
    check for palindromes.") ( (palindrome xs) == (reverse xs == xs) )
```

The input `xs` must be captured with a dependent contract before it can be referred to. This means that we are unable to derive a contract useful for contract inference.

The only QuickCheck properties that are of use to derive a contract from, are properties that only refer to the result of the function. For these kinds of properties, we can construct a contract of the following form:

```
PropInfo (λresult → ...)
```

As you can see, we are severely limited in the the range of properties that we can use. For this reason, we do not develop a contract deriving mechanism for QuickCheck properties and leave it as future work.

Chapter 5

Results and future work

This chapter presents the results of our work in section 5.1 in the form of several comparisons between the feedback provided by Ask-Elle and the feedback provided by our system given the same input. The rest of the chapter discusses interesting avenues for future work.

5.1 Results

Originally, our work was to be integrated into the Ask-Elle programming tutor, but time constraints have prevented this. Nonetheless, it is possible to emulate the effect our new source of feedback by taking student input (comprised of source code) and system output (comprised of QuickCheck counterexamples) stored by the Ask-Elle system and using the input and output to generate our own feedback (if any):

1. First, we create a contract derived from the QuickCheck properties that were used to generate the counterexample and place the counterexample in the contract as described by Stutterheim's thesis in subsection 3.5.1, *"Eliminating dependent contracts"*.
2. This contract is given to the contract environment Γ and used during contract inference, generating substitutions that may place the counterexample-laden contract at function applications.
3. Then, we run the student's input through the code generation system to generate contracted code.

4. We apply the contracted version of the code to the counterexample.
5. A contract violation occurs at a specific function application, providing detailed feedback.

Automation of the first step is discussed in subsection 4.8.4. The rest of the steps are not as troublesome to automate.

5.1.1 Strengths and weaknesses

We have performed several tests to determine the efficacy of our feedback compared to the original Ask-Elle feedback in identifying the source of a programming error.

Lack of function applications

Because contract violations only occur during assertion, code without any function applications does not benefit from our detailed feedback. For example, the following input will not generate more detailed feedback compared to the original Ask-Elle feedback:

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

Listing 5.1: Student input 1

```
You have not sorted correctly.
Counterexample arguments: [1,0]
```

Listing 5.2: Ask-Elle's response to input 1

In order to provide our own feedback, we go through the steps outlined above. Here is the original QuickCheck property:

```
prop_Main = λxs → prop_Elems xs .&&. prop_Model xs

prop_Elems = λxs → whenFail (putMsg "Input and output list do not contain
     the same elements") (and $ map ((flip elem) xs) (myreverse xs))

prop_Model = λxs → whenFail (putMsg "You have not sorted correctly") (
     myreverse xs == reverse xs)
```

And our manually derived contract equivalent:

```
prop_Elems = PropInfo (λxs → (and $ map ((flip elem) xs) ([1,0]) ))
 (λ → mkErrorMsg p "Input and output list do not contain the same
     elements.")

prop_Model = PropInfo (λxs → ([1,0] == reverse xs))
 (λ → mkErrorMsg p "You have not sorted correctly.")

prop_Main = prop_Elems & prop_Model

myreverse_ctrt  = (true <@> true) >-> prop_Main
```

Then, we run the contract inference algorithm and generate contracted code:

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

__contracted_myreverse ctrt posinfo =
    assertPos "At the application of the function 'myreverse'" (
        generatePositionData posinfo) ctrt funs
    where funs = (fun (λ__x01 → __final_myreverse __x01))

__app_myreverse ctrt posinfo (posa,a) =
    (appParam (__contracted_myreverse ctrt posinfo) (show a ++
        generatePositionData posa) a)
```

Here, the problem becomes apparent: because there are no function applications in the original code, no contracting has been done whatsoever! Of course, we can manually apply the contract with bogus information:

```
__app_myreverse myrevers_ctrt (Just (0,0)) ((Just (1,1)),[1,0])
```

Naturally, the feedback is not very useful:

> *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 'myreverse' at line number 0, column number 0. The result of this function does not fulfill the following property: You have not sorted correctly.*

Limitations of QuickCheck properties

In general, the QuickCheck properties used to describe the correct behaviour of the exercises in Ask-Elle are troublesome to correctly translate to a contract and often provide scarce information in their feedback, if any is present at all.

The QuickCheck property for the `repli` exercise, for example, provides no textual feedback whatsoever:

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

Furthermore, translation of this property to a contract is troublesome because of the need for a dependent contract to capture `n`, which Stutterheim illustrated to be unusable for this form of contract inference.

Defining custom contracts

In order to get the most out of the feedback potential that contract inference provides, contracts should be tailored to each exercise. Of course, the validation code must be shared with QuickCheck.

Let us assume the student has just finished an insertion sort exercise, but a programming error has crept in:

```
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
```

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]`.

The tailored contract that is given to the contract inference algorithm is as follows:

```
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]`, generating the following feedback:

> *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.*

Now we know that the issue lies with the `insert` function and not with our definition of `isort` or `foldr`. In this case, the extra information does not cut down the search space by a lot, but it is easy to imagine how this could be of great help in larger programs.

If we rectify the bug in the definition of `insert` (it should be `z : z' : zs'`) and introduce another by changing `z <= z'` to `z' <= z` and apply the counterexample again, we receive more detailed feedback:

> *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.*

Now, we are brought closer to the bug, indicating something apparently wrong with the snippet `z' : zs'`: one of the elements is causing the list to no longer be ordered. Unfortunately, we cannot get closer to the problem, as the contract inferred for the function application `(<=) z' z` is the identity contract `(true ⟼ (true ⟼ true))`. Nonetheless, the detail of this feedback is much greater than that of Ask-Elle's combination of a counterexample and a description of the violated property, our system displaying the violation as close to the source as it is allowed by the definition of the contract defined for the exercise.

5.2 Integration with the Ask-Elle programming tutor

Full integration of our work with the Ask-Elle programming tutor is not yet complete. Our system is somewhat integrated with the Ask-Elle programming tutor, primarily for its syntax. Full integration requires modifying the QuickCheck functionality in Ask-Elle to pass its counterexample to a function that constructs a contract used for inference. Stutterheim proposes a way to capture the counterexamples generated by QuickCheck in the future work chapter of his thesis for this purpose: by wrapping the `quickCheck` function and storing the result in an `IORef`, it is possible to capture the counterexample and read it from the `IORef` when it is needed.

Automatic translation of QuickCheck properties to contracts may fail to make use of the capabilities contracts offer, as contracts derived from QuickCheck properties can only say something about the *results*, and not about the *inputs*.

Manually constructing a contract that uses the same validation code as the QuickCheck properties do can offer a greater contracting space. The configuration files for exercises must be modified to include these contracts.

Furthermore, it will be necessary to run the code generation on the Prelude and any other modules that are used by exercises defined in Ask-Elle, so that the final contracts for all those functions are available for contract inference. Just as Ask-Elle maintains a static list of all the types of the Prelude, a static list of all contracts could be made available, too.

For those interested, *decoupling* our system from the Ask-Elle programming tutor is also feasible. Our use of a type source tightly couples our system with the Helium compiler, but if a source of types can be found for a new language such as Haskell, porting it over would be a realistic goal. Replacing the Ask-Elle syntax with the Helium syntax is also feasible, but will require some refactoring of the code generation system (mostly renaming).

5.3 Adding support for user-defined datatypes to the initial contract generation algorithm

As we have seen in subsection 4.4.2, contracts can be derived from types that are useful in contract inference. However, only a few types are currently supported by the conversion function. Custom data constructors must be manually added. A typeclass does not solve the need for manually adding conversions for new data constructors. Being able to look up data types instead of manually adding them is a more interesting idea. It is possible to query the Helium compiler for a list of all available type constructors, functions, value constructors and operators in a program. The Ask-Elle programming tutor also has a static list available in the file `Domain\FP\HeliumImportEnvs.hs`. Both sources could be used to look up unknown type constructors, fetch their type, and construct a fitting contract.

5.4 Runtime translation of intermediate contract language

Subsection 4.7.2 proposes splitting up generated substitutions into a bottom-up list and placing the other top-down substitutions into a top-down list unique to the node where unification took place. This change makes it possible to correct a limitation present in Stutterheim's original contract inference algorithm.

However, application of the substitutions and translation of the final contracts must then be done at run-time.

Several ideas were tested to perform this translation:

- **Add a phantom type to the data type representing the intermediate contract language.** The UUAGC system currently does not support this.
- **Use a typeclass.** This approach generates a similar problem to the one that we are trying to solve.
- **Use a typeclass + type synonyms.** Requires a lot of extra instances to properly work. Perhaps interesting when fully automated.
- **Use a typeclass + newtypes.** The most promising avenue of research.
- **Code duplication.** The nuclear approach.

5.4.1 Add a phantom type to the data type representing the intermediate contract language

The data type that represents the intermediate contract language is a data type automatically generated by the Utrecht University Attribute Grammar Compiler (UUAGC). Adding a phantom type to it would make conversion trivial. Unfortunately, an option for adding phantom types to a UUAGC-defined data type could not be found. Even if phantom types were supported, the generation code would have had to have been rewritten to use a nested tuple for the environment contract instead of a regular list, or use another form of heterogeneous lists.

5.4.2 Use a typeclass

```
class ConvertContract a where
 convert :: Contract → (Contract a)

instance ConvertContract a where
 convert (CVar s _) = true
 convert (CProp s) = searchEnvTuple s envTuple

instance ConvertContract (a ⇀ b) where
 convert (CArr c1 c2) = (⇀) (convert c1) (convert c2)

instance Bifunctor f ⇒ ConvertContract (f a b) where
 convert (CBifunctor o il ir) = (convert o) <@@> (convert il,convert ir)

instance Functor f ⇒ ConvertContract (f a) where
 convert (CFunctor o i) = (convert o) <@> (convert i)
```

```
testEnvTuple
  :: (Functor f, Ord d, Ord b, RealFrac e) ⇒
     ((String, Contract (f c ↣ [d])),
      ((String, Contract a),
       ((String, Contract [b]),
        ((String, Contract Char),
         ((String, Contract e), ([a], Contract f))))))
testEnvTuple = ( ("isort",sortCtrt), ( ("ctrue",true), ( ("ord",ord), ( ("
    isChar_prop",isChar_prop), ( ("isInt_prop", isInt_prop), ([], true) )
     ) ) ) )
```

Listing 5.3: Attempt at using a typeclass to recover the extra type information.

Because of the different phantom types, a nested tuple was needed to house the property environment, but the tuple lookup function refused to typecheck, as the typechecker tried to unify the different contracts. Additionally, this solution suffers from the need for a type annotation so the compiler knows which instance to select. Statically generating such an annotation would have been perfectly feasible, although it would have suffered from the same issue as the one we were trying to solve by doing the translation at run-time.

5.4.3 Use a typeclass + type synonyms

What about using type synonyms and the aforementioned typeclass?

```
instance ConvertContract a where
  convert (CVar s _) = true
type IsIntProp a = a
instance RealFrac (IsIntProp a) ⇒ ConvertContract (IsIntProp a) where
  convert (CProp p) | p == "isInt_prop" = isInt_prop
```

Listing 5.4: Attempt at using type synonyms to choose the correct instances.

Unfortunately, when deciding an instance, any constraints are ignored, and because type synonyms are transparent to the compiler, these instances fully overlap. Simon Peyton-Jones and Oleg Kiselyov do describe a way to use overlapping instances and functional dependencies to achieve this [14], but it requires a lot of extra work for every property. Automatic generation of these instances may be an interesting avenue.

5.4.4 Use a typeclass + newtypes

```
newtype IsIntProp a = IsIntProp a
instance RealFrac (IsIntProp a) ⇒ ConvertContract (IsIntProp a) where
  convert (CProp p) | p == "isInt_prop" = isInt_prop
```

Listing 5.5: Attempt at using newtypes to choose the correct instances.

This seems to work, but it requires wrapping and unwrapping of values, which is incompatible with the generated code. Recall the contracted version of `f`:

```
__app_f ctrt x = appParam (__contracted_f ctrt) x
__contracted_f ctrt = assertPos "f" ctrt funs
    where
        funs = (fun (λx → __final_f x))
__final_f x = __app_g (c1 ↣ (c2 <@> c1)) x
```

The value `x` would have to be unwrapped before it could be passed to `__final_f`, and these unwrapping functions would all have a different name. A typeclass could be used to provide a singular function to wrap and unwrap newtypes. Placing the generated code together with this typeclass in a single module and running it with GHC makes this the most interesting avenue to investigate.

5.4.5 Code duplication

A final "solution" that would not require the use of typeclasses, would have been to generate a monomorphic set of contracted functions for every single function application. Naturally, this set would need to contain every single function that is called when evaluating that particular function application; a possibly daunting amount. Importing other modules makes this solution even less feasible, as functions from those modules would also need to be included in the set. Because of the multitude of issues, this solution was disregarded early in the process.

5.5 More efficient algorithm for finding final contracts

The algorithm to find a final contract uses a brute-force search, going through a substitution list up $(n+1)$ times, where n is the number of substitutions in the list. This algorithm may become a bottleneck when it has to trudge through a very large bottom-up substitution list. The optional addition of top-down substitution lists is not as troublesome, because they contain only substitutions that are relevant to that node.

The current algorithm goes as follows:

Given a contract variable c_{orig} and a list of substitutions s, set c_i to c_{orig}. While you have not reached the end of s, search for a substitution that goes from c_i to c_j. If you encounter such a substitution, set c_i to c_j. When the end of s has

been reached, check if $c_{orig} = c_i$. If yes, return c_{orig} as the result. Otherwise, recurse with c_j and s.

Proving that this algorithm always terminates is easy. When applying a list of substitutions to a contract c, there are two possible results: either the result is once again c, or it is c'. In the first case, we are done. In the second, we must recurse.

The question is, is it possible to construct a substitution loop in which we always recurse and never terminate? The answer is no.

The proof is as follows: assume a substitution list s that only contains the substitution $\boxed{\texttt{c} \mapsto \texttt{c}'}$. When applying s to c, we obtain c'. Recursing, c' is applied to s again. This time, c' remains unchanged, so we are done. Let us extend s with substitution $\boxed{\texttt{c}' \mapsto \texttt{c}}$. We apply s to c, obtaining c itself, because the substitutions cancel each other out. So, we are done. If we apply s to c', we obtain c. Recursing, we end up in the previous case, so we are done as well. Applying a substitution list s that does not contain any substitutions that involve c will naturally result in c again, making all cases accounted for.

5.6 Adding support for partially applied function arguments to higher-order functions in *typed-contracts* code generation

At the moment, function arguments to higher-order functions are assumed not to be applied at all. These arguments are replaced by their *___contracted_*$\boxed{\texttt{X}}$ template. Of course, it is possible to pass partially applied functions to a higher-order function. To support such arguments, the generation code has to be modified to check if the function argument has been partially applied and, if so, replace it with its *___app_*$\boxed{\texttt{X}}$ template and wrap the applied values accordingly with `appParam`. Just as extensive feedback is generated when applying the function argument to a value in the *___contracted_*$\boxed{\texttt{X}}$ template for a higher-order function (see listing 4.10), so must similar feedback be generated for these arguments.

5.7 Using paramorphisms to tackle dependent contracts

Johan Jeuring provided an excellent idea to be able to run contract inference on some catamorphisms with dependent contracts by converting them to a paramorphism, which is a catamorphism that provides its function argument with the original input as well:

```
para  :: (a → [a] → b → b) → b → [a] → b
foldr :: (a →       b → b) → b → [a] → b

para  c n (x : xs) = c x xs (para c n xs)
foldr c n (x : xs) = c x    (foldr c n xs)
para  c n []       = n
foldr c n []       = n
```

Source: http://stackoverflow.com/questions/13317242/what-are-paramorphisms

This functionality is of great use for dependent contracts that need access to the original argument, as can be seen in the following example code (courtesy of Johan Jeuring):

```
-- GList      ::  Contract a → ([a] → Bool) → Contract [a]
sortc =  GList int (const True) ↣ λxs → GList int (λr → nonDesc r &&
    isPerm xs r)
isPerm xs ys = null (xs \\ ys)

sort_perm' :: Ord a ⇒ [a] → [a]
sort_perm' =  foldr insert_notPerm []
insert_notPerm ::  Ord a ⇒ a → [a] → [a]
insert_notPerm x []                  =  [x]
insert_notPerm x (y:ys)  |  x <= y     =  x:y:ys
                         |  otherwise  =  insert_notPerm x ys

sort_permWrong        =  foldr insert_notPerm []
sort_permNotViolated  =  assert sortc sort_permWrong [3,2,4]
```

In the above example the input, `xs`, is needed in the `sortc` contract to be able to express the permutation property, but this introduces a dependent contract, which means the contract cannot be used for contract inference.

This is where the paramorphism comes in, as it provides that `xs` value. The contract for the `para` function is, then:

```
parac
  :: Contract a
     → ([a] → Bool)
     → ([a] → Contract b)
     → Contract ((a → [a] → b → b) → b → [a] → b)
parac a p b =   (a ↣ λx → GList a p ↣ λxs → b xs ≻- b (x:xs))
              ≻-  b []
              ≻-  GList a p
              ↣ λxs -> b xs

psort  =  para (λx xs ys → insert_correct x ys) []
```

Unifying `sortc` and `parac` gives us:

```
GList a   p              ↣ λxs → b xs
GList int (const True)   ↣ λxs → GList int (λr → nonDesc r && isPerm xs
    r)

a   :=  int
p   :=  const True
b   :=  λxs → GList int (λr → nonDesc r && isPerm xs r)
```

Applying these substitutions, we see that `bT []` will get the contract:

```
GList int (λr → nonDesc r && isPerm [] r)
```

And the contract for `insert` becomes:

```
insertc'  =    int
          ↣ λx → GList int (const True)
          ↣ λxs → GList int (λr → nonDesc r && isPerm xs r)
          >-   GList int (λr → nonDesc r && isPerm (x:xs) r)
cinsert_notPerm_violated = assert insertc' (λx xs ys → insert_notPerm x
    ys) 3 [2,4] [2,4]
```

Note that by using a paramorphism instead of a catamorphism, `insert` is supplied with an argument it does not require, so it is wrapped in a lambda function and the superfluous argument is ignored.

How do we relate this contracted paramorphism to the original user's `foldr` call? By defining `foldr` as a paramorphism, which is quite simple:

```
pfoldr f e = para (λx xs r → f x r) e
sort_perm''         =  pfoldr insert_notPerm []
sort_permViolated' =  assert sortc' sort_perm'' [3,2,4]
```

In short, this implies it is possible to convert some functions that use catamorphisms with dependent contracts, into paramorphisms with non-dependent contracts.

Some feedback problems do pop up: how do we meaningfully inform the user of a violated contract in `(λx xs r → f x r)` ? And what if the `para` contract is violated, how do we report that? It is possible to modify the feedback messages in contract generation to "cloak" the use of a paramorphism, but this has not yet been investigated further.

Chapter 6

Conclusion

The main objective of this thesis was the extension of Stutterheim's contract inference algorithm to the Helium-like Ask-Elle syntax and the construction of a code generation system with the goal to integrate it into the tutor and provide more detailed feedback to students.

To achieve this, a framework was constructed that links the contract inference algorithm with the code generation system. Input to this framework consists of a program written in the Ask-Elle syntax, a slightly simplified version of the Helium functional programming language. The framework first applies several transformations to the input, removing syntactic sugar and similar simplifications before it is passed to the contract inference algorithm and code generation system.

Our contract inference algorithm Algorithm $\mathcal{CHW}$ is based on Stutterheim's Algorithm $\mathcal{CW}$. Unlike $\mathcal{CW}$, $\mathcal{CHW}$ uses a type source to simplify contract inference and focus on the inference uniquely required by contracts.

Stutterheim's code contained a proof-of-concept code generation system. We expanded this system considerably to create a framework for generating code for the *typed-contracts* contract library. Many elements of the code generation system are library-agnostic, so generation interfaces for other contract libraries can be added with nearly no code duplication.

We identified a limitation of the generated code due to the way substitutions are collected and applied in Stutterheim's proof-of-concept code generation system. A solution was devised and implemented in our system, but the last step, a run-time translation from the contract's data type used in Algorithm $\mathcal{CHW}$ to the contract's data type of the targeted library, proved to be harder than expected.

We listed our attempts to this problem and highlighted the more promising candidates. In the case that this translation problem is solved, the rest of our solution can be reactivated with some minor modifications to the substitution code. At the moment, the limitation is still in effect.

Full integration with the Ask-Elle programming tutor was deferred as future work. Through several examples, we showed how the feedback of our framework was equally or more detailed than the feedback of Ask-Elle when provided the same inputs. Especially large programs benefit the most from our improved feedback, cutting down the programming error search space down to a single function and sometimes a single function application. If possible, our feedback includes the value that caused the contract violation, a feature that is especially useful in recursive functions.

Decoupling our framework from Ask-Elle and making it available for the Helium language is also a possibility. Replacing the Ask-Elle syntax with that of Helium would require some refactoring, but no significant changes to the framework would have to be made. The result would be an automatic contract annotation framework for Helium, something that most certainly deserves further inquiry and research.

Appendices

Appendix A

Modified Ask-Elle syntax

```
-- | A Haskell source module
data ModuleR
  | ModuleR
      name      :: MaybeNameR
      body      :: BodyR
      range     :: RangeR

data BodyR
  | BHoleR
      id        :: HoleIDR
      range     :: RangeR
  | BodyR
      decls     :: DeclsR
      range     :: RangeR

-- | Declarations
data DeclR
  | DHoleR
      id        :: HoleIDR
      range     :: RangeR
  | DEmptyR
      range     :: RangeR
  | DFunBindsR
      funbinds  :: FunBindsR
      range     :: RangeR
  | DPatBindR
      pat       :: PatR
      rhs       :: RhsR
      range     :: RangeR

type DeclsR = [DeclR]

-- | Expressions
data ExprR
  | HoleR
      id        :: HoleIDR
      range     :: RangeR
  | FeedbackR
      feedback  :: String
      expr      :: ExprR
      range     :: RangeR
  | MustUseR
      expr      :: ExprR
      range     :: RangeR
  | CaseR
      expr      :: ExprR
      alts      :: AltsR
      range     :: RangeR
  | ConR
      name      :: NameR
      range     :: RangeR
  | IfR
      cond      :: ExprR
      then      :: ExprR
      else      :: ExprR
      range     :: RangeR
  | InfixAppR
      left      :: MaybeExprR
      op        :: ExprR
      right     :: MaybeExprR
      range     :: RangeR
  | LambdaR
      pats      :: PatsR
      expr      :: ExprR
      range     :: RangeR
  | LetR
      decls     :: DeclsR
      expr      :: ExprR
      range     :: RangeR
  | LitR
      lit       :: LiteralR
      range     :: RangeR
  | AppR
      fun       :: ExprR
      args      :: ExprsR
      range     :: RangeR
  | ParenR
      expr      :: ExprR
      range     :: RangeR
  | TupleR
      exprs     :: TupleExprR
      range     :: RangeR
  | VarR
      name      :: NameR
```

```
      range     :: RangeR
    | EnumR
      from      :: ExprR
      then      :: MaybeExprR
      to        :: MaybeExprR
      range     :: RangeR
    | ListR
      exprs     :: ListExprR
      range     :: RangeR
    | NegR
      expr      :: ExprR
      range     :: RangeR

type ExprsR = [ExprR]
type ListExprR = [ExprR]
type TupleExprR = [ExprR]

data MaybeExprR
    | NoExprR
    | JustExprR
      expr      :: ExprR

-- | Alternatives
data AltR
    | AHoleR
      id        :: HoleIDR
      range     :: RangeR
    | AltR
      feedback  :: {Maybe String}
      pat       :: PatR
      rhs       :: RhsR
      range     :: RangeR
    | AltEmptyR
      range     :: RangeR

type AltsR = [AltR]

-- | Function bindings
data FunBindR
    | FBHoleR
      id        :: HoleIDR
      range     :: RangeR
    | FunBindR
      feedback  :: {Maybe String}
      name      :: NameR
      pats      :: PatsR
      rhs       :: RhsR
      range     :: RangeR

type FunBindsR = [FunBindR]

-- | Guarded expressions
data GuardedExprR
    | GExprR
      guard     :: ExprR
      expr      :: ExprR
      range     :: RangeR

type GuardedExprsR = [GuardedExprR]

-- | Literal values
data LiteralR
    | LCharR
      val       :: Char
      range     :: RangeR
    | LFloatR
      val       :: Float
      range     :: RangeR
    | LIntR
      val       :: Int
      range     :: RangeR
    | LStringR
      val       :: String
      range     :: RangeR

-- | Names
data NameR
    | IdentR
      name      :: String
      range     :: RangeR
    | OperatorR
      name      :: String
      range     :: RangeR
    | SpecialR
      name      :: String
      range     :: RangeR

data MaybeNameR
    | NoNameR
    | JustNameR
      name      :: NameR

-- | Patterns
data PatR
    | PHoleR
      id        :: HoleIDR
      range     :: RangeR
    | PConR
      name      :: NameR
      pats      :: PatsR
      range     :: RangeR
    | PInfixConR
      left      :: PatR
      name      :: NameR
      right     :: PatR
      range     :: RangeR
    | PListR
      pats      :: ListPatsR
      range     :: RangeR
    | PLitR
      lit       :: LiteralR
      range     :: RangeR
    | PParenR
      pat       :: PatR
      range     :: RangeR
    | PTupleR
      pats      :: TuplePatsR
      range     :: RangeR
    | PVarR
      name      :: NameR
      range     :: RangeR
    | PAsR
      name      :: NameR
      pat       :: PatR
      range     :: RangeR
    | PWildcardR
      range     :: RangeR

type PatsR = [PatR]
type ListPatsR = [PatR]
type TuplePatsR = [PatR]

-- | Right hand side
data RhsR
    | RhsR
      expr      :: ExprR
```

```
      where     :: DeclsR
      range     :: RangeR
    | GRhsR
      gexprs    :: GuardedExprsR
      where     :: DeclsR
      range     :: RangeR

data RangeR
    | RangeR
        rangestart :: PositionR
        rangestop  :: PositionR
```

```
data PositionR
    | PositionR
        filename  :: String
        line      :: Int
        column    :: Int
    | UnknownR
```

Listing A.1: Our modified Ask-Elle syntax.

Bibliography

[1] Olaf Chitil. Practical Typed Lazy Contracts. In *ICFP'12*, pages 1–16, July 2012.

[2] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ICFP '00*, pages 268–279. Chalmers University of Technology, 2000.

[3] Patrick Cousot, Radhia Cousot, Francesco Logozzo, and Michael Barnett. An abstract interpretation framework for refactoring with application to extract methods with contracts.

[4] L Damas and R Milner. Principal type-schemes for functional programs. In *POPL '82*, pages 207–212, 1982.

[5] Markus Degen, Peter Thiemann, and Stefan Wehr. True lies: Lazy contracts for lazy languages (faithfulness is better than laziness). In *4. Arbeitstagung Programmiersprachen (ATPS'09)*, Lübeck, Germany, October 2009.

[6] Christos Dimoulas and Matthias Felleisen. On contract satisfaction in a higher-order world. *ACM Trans. Program. Lang. Syst.*, 33(5):16:1–16:29, November 2011.

[7] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. *SIGPLAN Not.*, 37(9):48–59, September 2002.

[8] Alex Gerdes, Bastiaan Heeren, and Johan Jeuring. Constructing strategies for programming. In José A. Moinhos Cordeiro, Boris Shishkov, Alexander Verbraeck, and Markus Helfert, editors, *CSEDU (1)*, pages 65–72. INSTICC Press, 2009.

[9] Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts made manifest. *SIGPLAN Not.*, 45(1):353–364, January 2010.

[10] Ralf Hinze, Johan Jeuring, and Andres Löh. Typed contracts for functional programming. In *In FLOPS '06: Functional and Logic Programming: 8th International Symposium*, pages 208–225. Springer-Verlag, 2006.

[11] Ranjit Jhala, Rupak Majumdar, and Andrey Rybalchenko. Refinement type inference via abstract interpretation. *arXiv preprint arXiv:1004.2884*, 2010.

[12] Andrew Kennedy. Programming languages and dimensions. Technical Report 391, University of Cambridge, April 1996.

[13] B Meyer. Eiffel: A language and environment for software engineering. *Journal of Systems and Software*, 1988.

[14] Simon Peyton-Jones and Oleg Kiselyov. Ghc / advancedoverlap. *http://www.haskell.org/haskellwiki/GHC/AdvancedOverlap*, October 2013.

[15] Patrick M Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. *ACM SIGPLAN Notices*, 43(6):159–169, 2008.

[16] Patrick M Rondon, Niki Vazou, and Ranjit Jhala. Abstract refinement types. In *ESOP*, 2013.

[17] Jurriën Stutterheim. Contract inferencing for functional programs. Master's thesis, Utrecht University, January 2013.

[18] Tachio Terauchi. Dependent types from counterexamples. In *ACM Sigplan Notices*, volume 45, pages 119–130. ACM, 2010.

[19] Dimitrios Vytiniotis, Simon Peyton Jones, Dan Rosén, and Koen Claessen. HALO: Haskell to Logic through Denotational Semantics. In *POPL'13*, 2013.

[20] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP '09, pages 1–16, Berlin, Heidelberg, 2009. Springer-Verlag.

[21] Dana N Xu. Extended Static Checking for Haskell. In *Haskell'06*, pages 1–12, September 2006.

[22] Dana N. Xu, Simon Peyton Jones, and Koen Claessen. Static contract checking for haskell. *SIGPLAN Not.*, 44(1):41–52, January 2009.