Universiteit Utrecht

Department of Computing Science

MSc Thesis, ICA-3555003

Contract Inferencing for Functional Programs

Author:
Jurriën Stutterheim

Supervisor:
Prof. dr. J.T. Jeuring

January 17, 2013

Contents

List of Figures

Abstract

Ask-Elle is a programming tutor that allows students to learn Haskell at their own pace, without a teacher's direct supervision. To enable this, the tutor requires model solutions to be specified up front by a teacher, after which the tutor is able to guide the student towards one of these solutions. However, should the student deviate from the model solutions too much, the tutor can no longer give assistance. In an attempt to still provide the student with feedback once that happens, we propose using both QuickCheck and contracts to locate potential bugs in the student's program. Since we, in general, do not know anything about the structure of a student's solution, we propose and implement a method for inferring contracts for the student's program.

Chapter 1

Introduction

Last year's (mandatory) bachelor level course in Functional Programming at Utrecht University had over 200 bachelor students attending lectures and lab sessions. While a great turn-up by itself, it is unfortunately practically impossible to answer every student's questions during such busy lab sessions. Once the lab sessions come to an end, the students that did not get to ask all their questions won't be able to ask them until the next lab sessions. While the teachers and assistants in the functional programming course had their hands full with supporting all of the enrolled students, the number they had to deal with pales in comparison with the numbers achieved in a relatively new phenomenon, called MOOCs: Massive Open On-line Courses. In a single MOOC instance, several tens of thousands of students from all over the world can take part at the same time, usually free of charge.

As one can image, no single one teacher can assist that many students at the same time, even with a small army of student assistants. So, instead of being constantly guided by a professor or a teaching assistant, students need to be able to study independently. While even in small courses, independent studying is strongly encouraged, small courses have the advantage that students can ask teachers and assistants for help when they get stuck. When classes reach the size of the functional programming course at Utrecht University, or when they reach a MOOC-level number of students, this gets significantly more difficult. One solution may be to encourage students to do peer-reviews of the exercises and help each other out when stuck. In practice, however, social barriers and variable availability of fellow students may limit the amount of useful feedback a student can receive. Automated teaching tools, also known as tutors, can provide a solution to this problem. These tutors allow students to work at home at their own pace, and provide consistent feedback on the student's progress.

Gerdes et al. [11] have developed a tutor, called Ask-Elle, that assists students in doing exercises for a bachelor-level functional programming course. In this course, students learn the basics of functional programming in Haskell [18]. Students can choose from several exercises, taken from the "H-99: Ninety-Nine

Haskell Problems" exercise set[1]. After selecting an exercise, students are presented with a short problem description and the type signature of the function they should implement. For example, if a student selects the insertion-sort exercise, he may be presented with a description similar to the following:

> Write a function that sorts a list: $sort : Ord\ a \Rightarrow [a] \rightarrow [a]$. For example:
>
> ```
> Data.List> sort [3,2,6,8,1]
> [1,2,3,6,8]
> ```

An empty text field is presented and the student can start working on the exercise. If the student finds himself truly stuck, he can ask the tutor for a hint. Should the student still be stuck after receiving the hint, he can ask the tutor to give a tiny snippet of code to help him work towards a solution. Finally, the student arrives at a solution, either with or without help from the tutor.

Of course, the situation described above is an ideal situation where the student implements a known solution to the problem at hand. In practice, guiding a student towards a solution is quite a bit trickier. Before the tutor can give hints, or tell the student that he has successfully completed the exercises (or not), the tutor needs to know what constitutes a solution to that specific exercise. Since the tutor cannot come up with programs by itself, a teacher will need to input one or more so-called model solutions beforehand. The tutor can then parse the model solutions and device strategies [12, 9] for working towards these solutions. When a student requests a hint, the tutor parses the student's program, and tries to match it against the known model solutions, or variations thereof.

Unfortunately, there are limits to the tutor's abilities to detect whether a student's program is a variation of one of the model solutions. Once a student deviates too much from all known model solutions, the tutor cannot assist the student further. While it very well might be that the student has implemented a correct program, the tutor cannot verify this any more. It is this situation that this thesis will focus on. Can we still provide the student with useful feedback, even when he has significantly deviated from the model solutions? Can we do this even when their programs might not be completely implemented yet? If a student has a bug in his program, can we find the bug's location and point the student to it?

As it turns out, we can do so, to some extent. In this thesis, we develop a method for inferring *contracts* (for an introduction to contracts, see Section 2.2) for functions and their sub-expressions. These contracts can be used to approximately locate a bug. Combined with QuickCheck [3], we can automatically generate input that triggers a contract violation. Specifically, this thesis makes the following contributions:

[1] http://www.haskell.org/haskellwiki/99_Haskell_exercises

- We extend the Ask-Elle tutor to fall back on QuickCheck when strategies and model solutions fall short, even when programs still contain holes.
- We infer contracts for a function and all its sub-expressions.
- Using counter-examples from QuickCheck, contract inference, and program transformations, we locate the position of bugs in a program.

We introduce our solution in the following chapters: Chapter 2 provides the reader with background information on the various concepts used in this thesis, describes the problem we have attempted to solve, and motivates our proposed solution. Chapter 3 then continues with a formal and technical description of our solution, after which Chapter 4 concludes with a discussion of the results and future work.

Chapter 2

Background

In this chapter we provide the reader with some background-information on the various concepts that are used throughout this thesis, and we clarify the problems that we have attempted to solve.

2.1 Ask-Elle, strategies, and QuickCheck

As mentioned in the introduction, the work we present here is motivated by the Ask-Elle programming tutor; a programming tutor that allows students to work on Haskell programming exercises at their own pace, without direct supervision of a teacher, while still being able to receive feedback on their progress. Enabling students to work by themselves in this way is more challenging than it may sound. Not only does the tutor need to be sophisticated enough to know when the student has finished an exercise, it also needs to be able to give the student feedback when he gets stuck. Enabling this in Ask-Elle is done by introducing *strategies* [9, 11, 10]. When creating an exercise, all a teacher has to do is enter one or more model solutions, which, in their simplest form, are just Haskell implementations of possible solutions to the exercise in question. These model solutions are parsed by the tutor and compiled into strategies. Using these strategies, the tutor can identify whether a student is working towards one of the model solutions, even if the variable names are different from the model solution, or if the program structure is different.

Despite the fact that the tutor's strategy system is sophisticated enough to recognise programs that deviate from a model solution, there are limits to this ability. Another situation where the use of strategies may fall short is when not all possible model solutions have been provided by the teacher, so the tutor cannot guide the student towards potentially good solutions. Practically, it is also impossible to come up with all possible model solutions, unless the exercise is trivial.

Rather than forcing a teacher to spend all his time trying to input all possible model solutions, we propose a different solution, which we have implemented in the tutor. In addition to specifying model solutions, we ask the teacher to also provide *properties* for the exercise. These properties are then used with QuickCheck [3] to use property-based testing on the submitted exercise. For example, for a sorting function, *sort*, we may define the following property:

```
prop_Sort : Ord a ⇒ [a] → Bool
prop_Sort xs = isNonDesc (sort xs) ∧ isPermutation xs (sort xs)
  where isPermutation xs ys = xs ∈ permutations ys
        isNonDesc (x :: y :: ys) = x ⩽ y ∧ isNonDesc (y :: ys)
        isNonDesc _              = True
```

QuickCheck then proceeds to generate *random* lists and applies the property function to them. After n (by default 100) successful tests, QuickCheck terminates and reports that, for these 100 random lists, *sort* adheres to the *prop_Sort* property. Should the *sort* function have a bug which causes it to violate the property, QuickCheck will produce a *counter-example*. It does by taking the randomly generated values and *shrinking* them. For example, suppose QuickCheck generates a list $[2, 0, 1]$, which causes the property to fail. It will then try to to see if the property still fails with the smaller list $[2, 0]$. If it does, it will try again with the even smaller list[1] $[\,]$. Should that succeed, it will return the counter-example $[2, 0]$ and show it to the user.

Using QuickCheck, we can give the student some feedback, even though the tutor is no longer able to use strategies to give exact feedback. If all tests succeed, the tutor can inform the student that, even though it is not entirely certain[2] that the implementation is correct, it certainly looks like the student is doing the right thing. Should the property fail, then the student is presented with a concrete counter-example, helping him to manually debug the program code.

While the addition of QuickCheck is certainly an improvement over having no feedback at all, it is still a sub-ideal way to give the student feedback. After all, the only lead a student has to find out *where* the bug in his program is, is the counter-example. Can we do better? It turns out that we can, by using *contracts*, which will be introduced in the next section.

2.1.1 Holes

Before continuing with a section about contracts, we need to answer one more question. Ask-Elle has the ability to analyse a student's incomplete program and provide feedback on the student's progress, allowing the student to incrementally develop his program. For this to work, the student needs to explicitly indicate the *holes* in the program. To incrementally implement *map*, for example, we might go through the following steps, in which ? indicates a hole:

[1]In reality, QuickCheck generates several other intermediate lists, but we omit them for brevity.

[2]There is always the chance that a counter-example is found with the $n + 1$th randomly generated value. Strictly speaking this also holds true for finite data structures, because QuickCheck does not guarantee that all permutations are tested.

$$map = ?$$

$$map\ f\ xs = ?$$

$$\begin{aligned} &map\ f\ [\,] &&= ?\\ &map\ f\ (x :: xs) &&= ?\end{aligned}$$

$$\begin{aligned} &map\ f\ [\,] &&= [\,]\\ &map\ f\ (x :: xs) &&= ? :: ?\end{aligned}$$

$$\begin{aligned} &map\ f\ [\,] &&= [\,]\\ &map\ f\ (x :: xs) &&= f\ x :: map\ f\ xs\end{aligned}$$

But what happens if we want to run QuickCheck on the second-last case? Clearly it is incomplete. QuickCheck supports a special *discard* exception since version 2.5. If this exception is thrown by the property under test, QuickCheck simply discards the test case and continues with a new one, until the given number of test cases has been reached. In the programming tutor we replace all holes with these *discard* exceptions, allowing us to QuickCheck students' programs, even if they are incomplete. For our specific example, this means that QuickCheck will need to generate n empty lists before the test succeeds.

2.2 Contracts

Contracts and the "design by contract" paradigm go back to at least Bertrand Meyer [16, 1], who coined the term in the 1980s in the context of his Eiffel [15] programming language. A contract specifies restrictions and gives guarantees for functions, much like a contract does so in real life between two parties. For example, a contract may specify that a function requires a natural number as argument, and that, provided a natural number is passed to the function, a natural number is returned as a result.

If a contract is violated, an exception is thrown which can include the location of the contract violation, and an indication as to which function is to blame for the contract violation. Adding contracts to programs therefore makes it easier to debug them, and incorporates tests in the program's code.

Contracts have been implemented in many imperative programming languages. Some languages, such as Eiffel, incorporate contracts as a language feature, while many of today's popular imperative programming languages support contracts as a library.

Contracts for functional programming languages are less widely applied. Nevertheless, contracts in functional languages are an active field of research. Findler and Felleisen [8] were the first to propose contracts for higher-order functions in Scheme [20]. Later, in 2006, Xu [22] introduced a static contract system, in which contracts can be written in Haskell and checked, using symbolic computation, at compile-time. Static contract systems like this share traits with dependently typed programming languages and theorem provers. Xu's work was followed by another paper in 2009, together with Peyton-Jones and Claessen [23]. An upcoming paper by Vytiniotis et al. [21] takes static contract checking a step further by translating contracts into first-order logic formulae and using an off-the-shelf theorem prover, to prove these formulae, rather than writing a custom contract system from scratch. On the other end of the spectrum, there is dynamic contract checking, which checks contracts at runtime. Hinze et al. [13] presented a library for dynamic contract checking in 2006 which was completely implemented in Haskell, without the need for modifications in the Haskell compiler. Based on the work by Hinze et al., Chitil [2] presents a newer library for typed lazy contracts, also implemented as a library in Haskell, which is claimed to preserve a program's lazy semantics, as opposed to the library by Hinze et al.

For this thesis we make use of the library by Hinze et al., called `typed-contracts`, whenever we need a concrete implementation of contracts. We chose this library mainly because we did not know of Chitil's work until after we had started implementing our ideas with the `typed-contracts` library. Concepts in this thesis should be portable to other libraries, and likely also to static contract systems. Our choice for a dynamic contract system is motivated by the desire to apply contracted functions to QuickCheck counter-examples, which can only be obtained at run-time, and by the fact that the dynamic contract systems are, at the moment of writing, readily usable after simply installing them, while the current static systems require installing experimental branches of GHC.

2.2.1 Asserting and implementing contracts

Contracts in the `typed-contracts` library have the same shape as the type of the function for which they are defined. For example, a contract for a function that goes from natural numbers to natural numbers is defined as follows:

$$nat \rightarrowtriangle nat$$

In order to assert a contract for a given function, the contract needs to be attached to, or wrapped around, the function first, which is done by the *assert* function. We borrow Hinze et al.'s example of a contracted definition of *head*, but change their naming convention to match our goals. When writing a contracted version of a function f, we rename the original function to f' and call the contracted version f.

$$\begin{aligned}&head : [\,a\,] \rightarrow a\\&head = assert\ (nonempty \rightarrowtriangle true)\ (\lambda xs \rightarrow head'\ xs)\end{aligned}$$

Since *head* is a partial function that is undefined for empty lists, we require the input list to be non-empty, which is ensured by the *nonempty* contract. From the type signature, we can see that the contracted *head* can be used as a drop-in replacement for the original *head*. Given the type-signature of *assert*, this is not surprising:

$$assert : Contract\ a \rightarrow (a \rightarrow a)$$

Given a contract, *assert* acts as a *partial identity*. It is partial, because the assertion raises an exception if the contract is violated. In the original paper by Hinze et al., the *Contract* type was implemented as a follows (using Generalised Algebraic Data Types; GADTs):

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

The *Prop* constructor takes a predicate and lifts it into a *Contract*, and the *And* constructor represents the conjunction of two contracts. The meaning of the remaining constructors is straight-forward. Note that both *Function* and *Pair* take a function as section argument, rather than simply a *Contract b*. This allows you to model *dependent function contracts* which allow you to use values from function arguments in the Definition of the contract. Section 3.5 discusses dependent contracts in the context of our work. For this thesis, we have added two new constructors for functors (types of kind $* \to *$) and bifunctors[3] (types of kind $* \to * \to *$) to the *Contract* GADT, which allow us to abstract over the *Pair* and *List* constructors:

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

In its simplest form, *assert*'s implementation is straight-forward and follows the *Contract* GADT closely:

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{assert}\ : \mathit{Contract}\ a \to (a \to a)} \\
\mathit{assert}\ (\mathit{Prop}\ p) & a & = \textbf{if}\ p\ a\ \textbf{then}\ a\ \textbf{else}\ \mathit{error}\ \texttt{"contract failed"} \\
\mathit{assert}\ (\mathit{Function}\ c_1\ c_2) & f & = (\lambda x' \to (\mathit{assert}\ (c_2\ x') \cdot f)\ x') \cdot \mathit{assert}\ c_1 \\
\mathit{assert}\ (\mathit{Pair}\ c_1\ c_2) & (a_1, a_2) & = (\lambda a_1' \to (a_1', \mathit{assert}\ (c_2\ a_1')\ a_2))\ (\mathit{assert}\ c_1\ a_1) \\
\mathit{assert}\ (\mathit{List}\ c) & \mathit{as} & = \mathit{map}\ (\mathit{assert}\ c)\ \mathit{as} \\
\mathit{assert}\ (\mathit{Functor}\ c) & \mathit{as} & = \mathit{fmap}\ (\mathit{assert}\ c)\ \mathit{as} \\
\mathit{assert}\ (\mathit{Bifunctor}\ c_1\ c_2) & \mathit{as} & = \mathit{bimap}\ (\mathit{assert}\ c_1)\ (\mathit{assert}\ c_2)\ \mathit{as} \\
\mathit{assert}\ (\mathit{And}\ c_1\ c_2) & a & = (\mathit{assert}\ c_2 \cdot \mathit{assert}\ c_1)\ a
\end{array}$$

[3]Haskell's base packages do not provide the *Bifunctor* type class. Instead, we use the class found in the *Data.Bifunctor* module in the `bifunctors` package found on Hackage. This choice is rather arbitrary, and other bifunctor implementations are easily used instead.

Asserting a *Prop* p simply applies the predicate p to the value a. If the predicate holds, *assert* acts as identity. If not, it raises an exception, signalling that the contract is violated. In case of a *Function* contract, we build up a new function in which the contract assertions are inlined. When this new function is applied to some argument, the contract for the function's domain is first asserted. Provided the first contract succeeds, the result value is used for the assertion of the co-domain contract. A *Pair* contract is similar in this respect, as it first asserts the contract for the left element, after which the left element's value is passed on to the right element's contract. Asserting *List*, *Functor*, and *Bifunctor* is straight-forward, as it just involves applying the corresponding homomorphisms. Lastly, the *And* contract requires that both contracts hold for the same value.

As Hinze et al. already indicate, and what the implementation also shows, is that the contracts for *Pair* and *List* follow a similar pattern. This observation is strengthened by the way *assert* is implemented for *Functor* and *Bifunctor*. Repeating what Hinze et al. already note in their paper, for some arbitrary container type T, we can define an assertion

$$\mathit{assert}\ (T\ c_1 \ldots c_n) = \mathit{mapT}\ (\mathit{assert}\ c_1) \ldots (\mathit{assert}\ c_n)$$

for some mapping function mapT for that specific type T.

assert's actual implementation is a bit more involved, as it needs to cope with proper blame-assignment in which the correct function is identified as violating a contract. For a more detailed discussion about blame assignment, see the original paper. The simplified implementation above suffices for this thesis.

To make defining contracts more intuitive, the contract library exposes several convenience functions:

$$\begin{array}{lll}
c_1 \twoheadrightarrow c_2 & = & \mathit{Function}\ c_1\ (\mathit{const}\ c_2) \\
c_1 \overset{d}{\longmapsto} c_2 & = & \mathit{Function}\ c_1\ c_2 \\
(\&) & = & \mathit{And} \\
c_1 \lessdot @ \gtrdot c_2 & = & c_1\ \&\ \mathit{Functor}\ c_2 \\
c_1 \lessdot @@ \gtrdot (c_2, c_3) & = & c_1\ \&\ \mathit{Bifunctor}\ c_2\ c_3
\end{array}$$

Of these convenience functions, the last two are the most interesting. Instead of being a simple wrapper for the *Functor* and *Bifunctor* contracts, they are a conjunction of the (bi)functor contracts and some other contract c_1. As an intuitive example of why this should be necessary, consider a list of integers. We could define a *List* contract that ensures that the list contains only natural numbers. However, knowing that a list contains only natural numbers does not say anything about the list as a whole. After all, we might want to require that the list be sorted, which can only be checked on the complete list. Our last two convenience functions combine the *outer contract* c_1, with which properties such as sorting can be captured, and *inner contract(s)*, with which properties like natural numbers can be captured. Capturing both outer and inner contracts at the same time is easy enough when defining contracts by hand, so having a convenience function available might not be a big improvement. However, as we will see in Chapter 3, these abstraction become very useful when inferring contracts.

With the *Contract* type and the convenience functions, we can define concrete contracts, the most extreme of which are the *true* and *false* contracts. The *true* contract always succeeds, while the *false* contract never succeeds and always raises an exception. An example where both are used is the *const* function, which always returns its first argument and always discards its second argument. Its contract can be defined as follows:

$$true \multimap false \multimap true$$

However, since *true* always succeeds, a perfectly valid, albeit less precice, contract for *const* is

$$true \multimap true \multimap true$$

and even just

$$true$$

in which case the *true* contract will just accept the *const* function as a whole.

We can also specify higher-order contracts, and using the (bi)functor contracts, we can also write contracts that closely follow the function's type. For example, for

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

we can define a contract

$$(c_1 \multimap c_2) \multimap (c_3 \lt\!@\!\gt c_1) \multimap (c_4 \lt\!@\!\gt c_2)$$

for some contracts c_1, c_2, c_3 and c_4. Since lists are functors, we choose to use the functor function here. The resulting contract looks very similar to *map*'s type. Contracts for universally quantified type variables behave very much like the type variables; the same type variable gets the same inner contract. However, we cannot, in general, say anything about the relationship between the two lists, so we assign them (potentially) different outer contracts.

2.2.2 Contracts in Ask-Elle

As mentioned in Section 2.1, QuickCheck is only able to give us a counterexample, and cannot tell us where in the program the property is being violated. Contracts, on the other hand, are ideally suited to locate the source of a property-violation, since they can be added to every single function in a program. In the context of Ask-Elle, however, we cannot expect the student to manually annotate the program with contracts, and since we are working in the situation where strategies can no longer help us, we cannot rely on strategies to annotate the program with contracts either. Luckily, all is not lost. What we do know is *which* function the student is currently trying to implement, even though we do not know *how* the student is implementing it. Since we know which function the student is implementing, we can, at the very least, specify a contract for the top-level function. Together with this top-level contract, we can then try to *infer* contracts for the rest of the program, much in the same way as types are being inferred by a compiler's type-checker in the presence of an explicit type signature. How this process works is detailed in Chapter 3. With a fully contracted program, and a QuickCheck counter-example, we are in a very good position to find the location of a bug. After all, we know for a fact that the counter-example is an input for which the function's property fails. Therefore, if a function is completely covered with contracts, applying it to the counter-example should lead us to the location of the bug.

We are not the first to propose inferring contracts. Dynamically inferring program invariants in imperative programming languages goes back to at least Ernst's dissertation [7] in 2000. Static invariant analysis goes back even further, to work by Cousot and Cousot [4] in 1977. One can even argue that Hoare's work on axiomatising computer programming [14] can be seen as one of the first forms of static contract inferencing. More recently, Cousot et al. [5] describe a method for inferring contracts for extracted methods in imperative languages. In between Hoare's original publication and Cousot's latest publication, a vast collection of work on deducing contracts for imperative programs is available, which mostly focusses on use-cases in C or Java. Contract inference for functional programs is significantly less well explored, however. To our knowledge, this thesis is the first to describe contract inferencing for (modern) functional languages.

Chapter 3

Contract inferencing

Contract inferencing is the process of automatically deducing contracts from a program's implementation. In many ways contract inferencing is similar, and in some places identical, to type inferencing as described by Damas and Milner [6], both conceptually and practically.

In contract inferencing, we want to achieve the following goals:

- 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

We have developed a contract system and an inferencing algorithm that is heavily based on Milner's Algorithm $\mathcal{W}$ [17], Damas and Milner's extension of this algorithm, and, like Damas and Milner's work, uses Robinson's unification algorithm [19]. Our contracting system and our inferencing algorithm, called Algorithm $\mathcal{CW}$, will be discussed in Section 3.4. The inferencing has some shortcomings, which will be discussed in Chapter 4.

3.1 The λ_c language

In order to explore the problem of contract inferencing, and develop our contract inferencing algorithm, we have created a small, simple let-polymorphic expression language called λ_c, presented in Figure 3.1. While this language is in no way suited for day-to-day programming, it features the main concepts of full-fledged functional programming languages, such as Haskell, hence we expect that the results from this thesis carry over to more mature languages.

In addition to the usual lambda calculus expressions, **let**-blocks, and **case**-blocks, the language has support for constants, built-in support for several datatypes, such as lists, *Maybe*, pairs, and *Either*, and binary operations. To simulate the programming tutor, the language also supports holes, denoted by a

$$
\begin{array}{rll}
\mathit{expr} ::= & x & \text{-- Variable} \\
\mid & \lambda \mathit{expr} \rightarrow \mathit{expr} & \text{-- Lambda abstraction} \\
\mid & \mathit{expr}\ \mathit{expr} & \text{-- Application} \\
\mid & \mathbf{let}\ \mathit{expr} = \mathit{expr}\ \mathbf{in}\ \mathit{expr} & \text{-- Let binding} \\
\mid & \mathbf{case}\ \mathit{expr}\ \mathbf{of} & \text{-- Case block} \\
& \quad \{\mathit{expr} \rightarrow \mathit{expr}\ (; \mathit{expr} \rightarrow \mathit{expr}){*}\} & \\
\mid & \mathit{const} & \text{-- Constants} \\
\mid & \mathit{expr} :: \mathit{expr} & \text{-- List cons constructor} \\
\mid & [\,] & \text{-- List nil constructor} \\
\mid & \mathit{Just}\ \mathit{expr} & \text{-- Maybe Just constructor} \\
\mid & \mathit{Nothing} & \text{-- Maybe Nothing constructor} \\
\mid & (\mathit{expr}, \mathit{expr}) & \text{-- Pair} \\
\mid & \mathit{Left}\ \mathit{expr} & \text{-- Either left constructor} \\
\mid & \mathit{Right}\ \mathit{expr} & \text{-- Either right constructor} \\
\mid & \mathit{expr} \oplus \mathit{expr} & \text{-- Binary operation} \\
\mid & ? & \text{-- Holes} \\
\mathit{const} ::= & n & \text{-- Integers} \\
\mid & b & \text{-- Booleans} \\
\mid & c & \text{-- Characters} \\
\mid & s & \text{-- Strings}
\end{array}
$$

Figure 3.1: Grammar for the expression language

question mark. With its basis in the lambda calculus, and due to its declarative nature, it is not hard to imagine that we can apply Algorithm $\mathcal{W}$ to infer types in this language. Section 3.4 will confirm that we can also infer contracts for this language.

3.2 Formalising the contract language

We define a contract language that is agnostic of a specific contract library. The grammar of this language is formalised in Figure 3.2. Since our implementation uses the contract library by Hinze et al. [13], the language is inspired by their notation and the notation of our additions to the library. Contracts for specific libraries can be generated from our language. In this thesis, we generate contracts for the library by Hinze et al., but we can also generate code for other libraries, such as the library by Chitil [2].

A contract is a user-defined concrete contract, a *true* contract which never fails, a $false$ contract which always fails, a contract for functions, which goes from a contract to a contract, or a contract for (bi)functors. We also explicitly add terminals for constants and data types, which serve as default contracts for the corresponding types. Lastly, we have contract schemes, with which we can

-- Contracts	
$c ::= \rho_\alpha$	-- User-defined concrete contract
$\mid\ true_\alpha$	-- true contract
$\mid\ false_\alpha$	-- false contract
$\mid\ c_\alpha \rightarrowtriangle c_\beta$	-- Function contracts
$\mid\ c_\alpha$ <@> c_β	-- Functor contracts
$\mid\ c_\alpha$ <@@> (c_β, c_γ)	-- Bifunctor contracts
$\mid\ int_\alpha$	-- Succeeds for all integers, and only for integers
$\mid\ bool_\alpha$	-- Succeeds for all booleans, and only for all booleans
$\mid\ char_\alpha$	-- Succeeds for all characters, and only for all characters
$\mid\ string_\alpha$	-- Succeeds for all strings, and only for all strings
$\mid\ list_\alpha$	-- Succeeds for all lists, and only for all lists
$\mid\ either_\alpha$	-- Succeeds for all *Either*s, and only for all *Either*s
$\mid\ maybe_\alpha$	-- Succeeds for all *Maybe*s, and only for all *Maybe*s
$\mid\ pair_\alpha$	-- Succeeds for all pairs, and only for all pairs
-- Contract schemes	
$\sigma ::= c$	-- Contract
$\mid\ \forall true_\alpha.\sigma$	-- Universal quantification for contract indices

Figure 3.2: Grammar for the contract language

universally quantify over *true* contracts. We will see that these contracts can be refined by unifying with more specific contracts.

Contracts for *true*, *false*, and the data types have an index α in order to distinguish between two instances of the same contract. We will omit the indices in our examples when doing so does not lead to ambiguity. Why we need indices will be discussed in Section 3.4.2. For now it suffices to know when two contracts are different, or stated otherwise, when they are equivalent:

Definition 1 (Equivalency of contracts). *$c_i \equiv d_j$ iff $c \equiv d$ and $i \equiv j$.*

3.3 Contract relations

In Haskell, a function's type statically guarantees that a function will be applied to a value of that type. Haskell's type system guarantees that a function f of type $Int \to Int$ is only applied to an integer in the range $[-2^{29}, 2^{29} - 1]$ on a 32 bit machine, and that it returns an integer in the same range as result. Any contract we infer for f must allow a subset of integers in that range as well. Since we know from the types that f ranges over integers, we can infer the contract

$$int_1 \rightarrowtriangle int_2$$

for it. If possible, however, we still want to refine this contract, since by itself it does not offer more guarantees than Haskell's type system; it always succeeds. We want to be able to make the contract more specific and allow a smaller subset of Haskell values. By making a contract more specific than the types, we enable it to fail assertion. For example, suppose we know that f does not range over all integers, but only over the natural numbers. We want to be able to make f's contract more specific and replace it with the contract

$$nat_1 \rightarrowtriangle nat_2$$

In order to know when we can make a contract more specific, we need to define relations between contracts. By regarding contracts as sets of Haskell values, we can do so. We formalise this idea in Definition 2.

Definition 2 (Semantics of contracts)**.** *The semantics of a contract c, written $[\![c]\!]$, is defined as the set of Haskell values for which it never fails assertion.*

From Definition 2 follows that $[\![true]\!]$ is the set of all Haskell values, since assertion never fails, and $[\![false]\!]$ is the empty set, since assertion always fails. The semantics of any contract is therefore a subset of $[\![true]\!]$ and a superset of $[\![false]\!]$. We formalise this thought in Proposition 1.

Proposition 1 (Contract relations with $true$ and $false$)**.** *For all contracts c, $[\![false]\!] \subseteq [\![c]\!] \subseteq [\![true]\!]$*

Proof. This follows from the definition of $[\![false]\!]$, $[\![true]\!]$ and $\subseteq$. □

Using these subset relations, we can refine contracts and make them more specific, increasing the chances that a contract assertion will fail, thereby locating a potential bug in the program. If $[\![c_1]\!] \subseteq [\![c_2]\!]$, then we can substitute $[\![c_1]\!]$ for $[\![c_2]\!]$ during refinement. With the subset relation we keep the property that if an assertion fails for c_2, it will also fail for c_1 (Proposition 2), and conversely that if assertion succeeds for c_1, it will also succeed for c_2 (Proposition 3).

Proposition 2 (Assertion fails for subset)**.** *For all contracts c_1, c_2, if $[\![c_1]\!] \subseteq [\![c_2]\!]$ and $assert\ c_2\ e = blame$, then $assert\ c_1\ e = blame$.*

Proof. By Definition 2 we can restate this in terms of sets: for all contracts c_1, c_2, if $[\![c_1]\!] \subseteq [\![c_2]\!]$ and $e \notin [\![c_2]\!]$, then $e \notin [\![c_1]\!]$. This follows from the set-theoretic definitions of $\subseteq$ and $\notin$. □

Proposition 3 (Assertion succeeds for superset)**.** *For all contracts c_1, c_2, if $[\![c_1]\!] \subseteq [\![c_2]\!]$ and $assert\ c_1\ e = e$, then $assert\ c_2\ e = e$.*

Proof. By Definition 2 we can restate this in terms of sets: for all contracts c_1, c_2, if $[\![c_1]\!] \subseteq [\![c_2]\!]$ and $e \in [\![c_1]\!]$, then $e \in [\![c_2]\!]$. This follows from the set-theoretic definitions of $\subseteq$ and $\in$. □

The semantics of function contracts also adheres to the subset relation. For example,

$$true_1 \twoheadrightarrow true_2$$

is the contract of all functions. Using Proposition 1, we can order it as follows

$$[\![false]\!] \subseteq [\![true_1 \twoheadrightarrow true_2]\!] \subseteq [\![true_3]\!]$$

Since not all Haskell values are functions and $[\![true_1 \twoheadrightarrow true_2]\!]$ is not empty, we can even make this a strict subset relation:

$$[\![false]\!] \subset [\![true_1 \twoheadrightarrow true_2]\!] \subset [\![true_3]\!]$$

Intuitively, it seems likely that we can use the subset relation for two function contracts as well. For example, if we have two contracts *nat* and *int*, for which holds that $[\![nat]\!] \subseteq [\![int]\!]$ (which follows from the conventional mathematical subset relation between natural numbers and integers), we can imagine that

$$[\![nat \twoheadrightarrow nat\ ]\!] \subseteq [\![int \twoheadrightarrow int]\!]$$

holds as well. We show that this is indeed the case in Proposition 4.

Proposition 4 (Superset relation for function contract semantics)**.** *For all contracts c_1, c_2, c_3, c_4, if $[\![c_1]\!] \subseteq [\![c_2]\!]$ and $[\![c_3]\!] \subseteq [\![c_4]\!]$, then $[\![c_1 \twoheadrightarrow c_3]\!] \subseteq\ [\![c_2 \twoheadrightarrow c_4]\!]$.*

Proof. By Proposition 3, we can reason that if $assert\ (c_1 \twoheadrightarrow c_3)\ f = f$ implies $assert\ (c_2 \twoheadrightarrow c_4)\ f = f$, then $[\![c_1 \twoheadrightarrow c_3]\!] \subseteq\ [\![c_2 \twoheadrightarrow c_4]\!]$. We show this by equational reasoning.

H1 $[\![c_1]\!] \subseteq [\![c_2]\!]$

H2 $[\![c_3]\!] \subseteq [\![c_4]\!]$

$$\begin{array}{ll}
 & assert\ (c_1 \twoheadrightarrow c_3)\ f = f \\
\equiv & \quad \{ \text{ Definition of } (\twoheadrightarrow) \text{ and } assert\ \} \\
 & \lambda x' \rightarrow (assert\ (const\ c_3\ x') \circ f)\ x' \circ assert\ c_1 = f \\
\equiv & \quad \{ \text{ Definition of } const\ \} \\
 & \lambda x' \rightarrow (assert\ c_3 \circ f)\ x' \circ assert\ c_1 = f \\
\Rightarrow & \quad \{ \text{ Apply H1 by Proposition 3 } \} \\
 & \lambda x' \rightarrow (assert\ c_3 \circ f)\ x' \circ assert\ c_2 = f \\
\Rightarrow & \quad \{ \text{ Apply H2 by Proposition 3 } \} \\
 & \lambda x' \rightarrow (assert\ c_4 \circ f)\ x' \circ assert\ c_2 = f \\
\equiv & \quad \{ \text{ Definition of } const\ \} \\
 & \lambda x' \rightarrow (assert\ (const\ c_4\ x') \circ f)\ x' \circ assert\ c_2 = f \\
\equiv & \quad \{ \text{ Definition of } assert \text{ and } (\twoheadrightarrow)\ \} \\
 & assert\ (c_2 \twoheadrightarrow c_4)\ f = f
\end{array}$$

□

A variation of the proof of Proposition 4 can be given by using Proposition 2 instead and by starting with $\mathit{assert}\ (c_2 \multimap c_4)\ f = \mathit{blame}$. For our example, we can now use Proposition 4 to refine the function contract

$$\mathit{int}_1 \multimap \mathit{int}_2$$

to

$$\mathit{nat}_1 \multimap \mathit{nat}_2$$

3.4 The contract system

This section describes a contracting system with which we can infer contracts from expressions. In this system, Γ represents a *contract environment* that maps variables to contracts and is defined as

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

where Γ is either the empty environment, or some environment Γ_1 extended by a mapping from some variable x to some contract c. We write $\Gamma(x) = c$ if the right-most binding for x in Γ maps x to c. For a contracting relation, we write $\Gamma \vdash e : c$ to denote that in environment Γ, expression e has contract c. We write $fc(\sigma)$ for the set of *true* contracts that appear free in contract scheme σ, and $fc(\Gamma)$ for the set of *true* contracts that appear free in codomain of Γ. In addition, we define two support functions, shown in Figure 3.3.

$$
\begin{array}{l}
\mathit{gen} : \mathit{Environment} \rightarrow \mathit{ContractScheme} \rightarrow \mathit{ContractScheme} \\
\mathit{gen}\ \Gamma\ c = \forall \mathit{true}_i. \ldots \forall \mathit{true}_n.c \\
\quad \textbf{where}\ \{\mathit{true}_i, \ldots, \mathit{true}_n\} = fc(c) \setminus fc(\Gamma) \\
\mathit{inst} : \mathit{ContractScheme} \rightarrow \mathit{ContractScheme} \\
\mathit{inst}\ (\forall \mathit{true}_i, \ldots, \forall \mathit{true}_n \,.\, c) = [\mathit{true}_i \mapsto \mathit{true}_{i'}] \ldots [\mathit{true}_n \mapsto \mathit{true}_{n'}]c \\
\quad \textbf{where}\ \mathit{true}_{i'}, \ldots, \mathit{true}_{n'}\ \mathit{are\ fresh}
\end{array}
$$

Figure 3.3: Generalization and instantiation functions

Starting from the top, *gen* generalises a contract by introducing universally quantified *true* contracts. Any *true* contract that occurs free in the contract and is not bound in the environment is quantified over. Instantiation, implemented by *inst*, removes the quantifiers and replaces all previously-bound *true* contracts with fresh ones. Both *gen* and *inst* are essentially the same as in Damas-Milner type inference, except we quantify over indexed *true* contracts, rather than type variables. Finally, we present the contracting rules in Figure 3.4.

$$\frac{\Gamma(x) = c}{\Gamma \vdash x : \mathit{inst}\ (c)}\ \textsc{C-Var} \qquad \frac{\Gamma[x \mapsto c_1] \vdash e : c_2}{\Gamma \vdash \lambda x \rightarrow e : c_1 \rightarrowtriangle c_2}\ \textsc{C-Lam}$$

$$\frac{\Gamma \vdash e_1 : c_2 \rightarrowtriangle c \qquad \Gamma \vdash e_2 : c_2}{\Gamma \vdash e_1\ e_2 : c}\ \textsc{C-App} \qquad \frac{\mathit{fresh}(i)}{\Gamma \vdash e : \mathit{true}_i}\ \textsc{C-True}$$

$$\frac{\Gamma[x \mapsto c_1] \vdash e_1 : c_1 \qquad \Gamma[x \mapsto \mathit{gen}_\Gamma(c_1)] \vdash e_2 : c}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : c}\ \textsc{C-Let}$$

$$\frac{\Gamma \vdash x : c \qquad \Gamma \vdash \mathit{xs} : \mathit{list} <@> c}{\Gamma \vdash (x :: \mathit{xs}) : \mathit{list} <@> c}\ \textsc{C-Cons}$$

$$\frac{\mathit{fresh}\ (i, j)}{\Gamma \vdash [\,] : \mathit{list}_i <@> \mathit{true}_j}\ \textsc{C-Nil}$$

$$\frac{\Gamma \vdash x : c \qquad \mathit{fresh}\ (i)}{\Gamma \vdash \mathit{Just}\ x : \mathit{maybe}_i <@> c}\ \textsc{C-Just}$$

$$\frac{\mathit{fresh}\ (i, j)}{\Gamma \vdash \mathit{Nothing} : \mathit{maybe}_i <@> \mathit{true}_j}\ \textsc{C-Nothing}$$

$$\frac{\Gamma \vdash x : c_1 \qquad \Gamma \vdash y : c_2 \qquad \mathit{fresh}\ (i)}{\Gamma \vdash (x, y) : \mathit{pair}_i <@@> (c_1, c_2)}\ \textsc{C-Pair}$$

$$\frac{\Gamma \vdash x : c_1 \qquad \mathit{fresh}\ (i, j)}{\Gamma \vdash \mathit{Left}\ x : \mathit{either}_i <@@> (c_1, \mathit{true}_j)}\ \textsc{C-EitherL}$$

$$\frac{\Gamma \vdash x : c_2 \qquad \mathit{fresh}\ (i, j)}{\Gamma \vdash \mathit{Right}\ x : \mathit{either}_i <@@> (\mathit{true}_j, c_2)}\ \textsc{C-EitherR} \qquad \frac{\mathit{fresh}\ (i)}{\Gamma \vdash\ ? : \mathit{true}_i}\ \textsc{C-Hole}$$

$$\frac{n \text{ is an integer} \qquad \mathit{fresh}\ (i)}{\Gamma \vdash n : \mathit{int}_i}\ \textsc{C-Integer}$$

$$\frac{b \text{ is a boolean} \qquad \mathit{fresh}\ (i)}{\Gamma \vdash b : \mathit{bool}_i}\ \textsc{C-Boolean}$$

$$\frac{c \text{ is a character} \qquad \mathit{fresh}\ (i)}{\Gamma \vdash c : \mathit{char}_i}\ \textsc{C-Character}$$

$$\frac{s \text{ is a string} \qquad \mathit{fresh}\ (i)}{\Gamma \vdash s : \mathit{string}_i}\ \textsc{C-String}$$

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

$$\frac{\Gamma \vdash e_1 : c_1 \qquad \Gamma \vdash e_2 : c_2 \qquad \Gamma \vdash \oplus{:}c_1 \rightarrowtriangle c_2 \rightarrowtriangle c_3}{\Gamma \vdash e_1 \oplus e_2 :: c_3}\ \textsc{C-BinOp}$$

Figure 3.4: Contracting rules

We will not discuss the rules for C-Var, C-Lam and C-App, because they are the same as in Damas and Milner's work. Not explicitly described by Damas and Milner, however, is our treatment of data types. After all, a data type can be converted into a lambda expression, after which the rules for the lambda calculus apply again. Our explicit modelling of functors and bifunctors in the contract language also requires us to explicitly specify contract rules for these contracts. In addition to the data types, the following paragraph will also expand on the rule for C-True and the rules for case blocks and binary operators.

Starting with C-True, we see that at any time we can create a *true* contract for an expression. After all, the *true* contract succeeds for all Haskell values, including functions. Continuing with with the case for C-Nil, we get a list functor contract with fresh indices. Since we know nothing about the contract of either the inner, or the outer contract, both contracts need to be fresh. The same holds for C-Nothing. In the case for C-Cons, we do know the contract for the element in the list. This element needs to have the same contract as the inner contract of the tail of the list in the same way that elements of a list need to have the same type. Hence, given that the element on top of the list has the same contract as the inner contract of the tail of the list, we can give it the same contract as for the tail of the list. For C-Just, we already know the contract for the variable inside the *Just*, but we need a fresh *just* contract for the outer contract. C-Pair, although it being a bifunctor, works similarly to C-Just, with the difference that it has two inner elements. The cases for C-EitherL and C-EitherR are more interesting. For either constructor, we can only know one of the inner contracts: either the left, or the right one. As a result, we need a fresh *true* contract for one of the inner contracts. Next, we have a case for a hole in a program, C-Hole. Since we cannot say anything about the contract of the hole, we just create a single fresh *true* contract. For each of the constants, such as integers, strings, characters, and booleans, we give a fresh type-specific contract. C-Case describes contract inference for case blocks with an arbitrary number of cases. Lastly, C-BinOp describes how we deal with binary operators.

3.4.1 Applying the contract rules

We present three examples to intuitively show how we can use our contracting rules from Figure 3.4 to deduce contracts from expressions. To save horizontal space on the page, we write t for the *true* contract.

const

$$\cfrac{\cfrac{\cfrac{\Gamma[x \mapsto t_1, y \mapsto t_2](x) = t_1}{\Gamma[x \mapsto t_1, y \mapsto t_2] \vdash x : t_1}}{\Gamma[x \mapsto t_1] \vdash \lambda y \to x : t_2 \rightarrowtriangle t_1}}{\Gamma \vdash \lambda x \to \lambda y \to x : t_1 \rightarrowtriangle t_2 \rightarrowtriangle t_1}$$

fix

$$\dfrac{\text{left subtree} \qquad \dfrac{\Gamma[fix \mapsto gen_\Gamma\ ((t \rightarrowtriangle t) \rightarrowtriangle t)](fix) = (t \rightarrowtriangle t) \rightarrowtriangle t}{\Gamma[fix \mapsto gen_\Gamma\ ((t \rightarrowtriangle t) \rightarrowtriangle t)] \vdash \mathit{fix} : (t \rightarrowtriangle t) \rightarrowtriangle t}}{\Gamma \vdash \textbf{let}\ \mathit{fix} = \lambda f \rightarrow f\ (\mathit{fix}\ f)\ \textbf{in}\ \mathit{fix} : (t \rightarrowtriangle t) \rightarrowtriangle t}$$

$$\dfrac{\dfrac{\dfrac{\text{left left subtree}}{\Gamma[fix \mapsto (t \rightarrowtriangle t) \rightarrowtriangle t, f \mapsto t \rightarrowtriangle t\ ] \vdash f\ (\mathit{fix}\ f) : t}}{\Gamma[fix \mapsto (t \rightarrowtriangle t) \rightarrowtriangle t] \vdash \lambda f \rightarrow f\ (\mathit{fix}\ f) : (t \rightarrowtriangle t) \rightarrowtriangle t}}{\text{left subtree}}$$

$$\dfrac{\dfrac{\dfrac{\Gamma[fix \mapsto (t \rightarrowtriangle t) \rightarrowtriangle t, f \mapsto t \rightarrowtriangle t\ ](f) = t \rightarrowtriangle t}{\Gamma[fix \mapsto (t \rightarrowtriangle t) \rightarrowtriangle t, f \mapsto t \rightarrowtriangle t\ ] \vdash f : t \rightarrowtriangle t} \qquad \dfrac{\text{right subtree}}{\Gamma[fix \mapsto (t \rightarrowtriangle t) \rightarrowtriangle t, f \mapsto t \rightarrowtriangle t\ ] \vdash \mathit{fix}\ f : t}}{\Gamma[fix \mapsto (t \rightarrowtriangle t) \rightarrowtriangle t, f \mapsto t \rightarrowtriangle t\ ] \vdash f\ (\mathit{fix}\ f) : t}}{\text{left left subtree}}$$

$$\dfrac{\dfrac{\dfrac{\Gamma[fix \mapsto (t \rightarrowtriangle t) \rightarrowtriangle t, f \mapsto t \rightarrowtriangle t\ ](fix) = (t \rightarrowtriangle t) \rightarrowtriangle t}{\Gamma[fix \mapsto (t \rightarrowtriangle t) \rightarrowtriangle t, f \mapsto t \rightarrowtriangle t\ ] \vdash \mathit{fix} : (t \rightarrowtriangle t) \rightarrowtriangle t} \qquad \text{right right subtree}}{\Gamma[fix \mapsto (t \rightarrowtriangle t) \rightarrowtriangle t, f \mapsto t \rightarrowtriangle t\ ] \vdash \mathit{fix}\ f : t}}{\text{right subtree}}$$

$$\dfrac{\dfrac{\Gamma[fix \mapsto (t \rightarrowtriangle t) \rightarrowtriangle t, f \mapsto t \rightarrowtriangle t\ ](f) = t \rightarrowtriangle t}{\Gamma[fix \mapsto (t \rightarrowtriangle t) \rightarrowtriangle t, f \mapsto t \rightarrowtriangle t\ ] \vdash f : t \rightarrowtriangle t}}{\text{right right subtree}}$$

null

$$\dfrac{\dfrac{\text{left subtree} \qquad \text{right subtree}}{\Gamma[xs \mapsto list_1 \mathbin{<\!@\!>} t_2\ ] \vdash \textbf{case}\ \mathit{xs}\ \textbf{of}\ \{[\,] \rightarrow \mathit{True}; (y :: ys) \rightarrow \mathit{False}\} : bool_3}}{\Gamma \vdash \lambda \mathit{xs} \rightarrow \textbf{case}\ \mathit{xs}\ \textbf{of}\ \{[\,] \rightarrow \mathit{True}; (y :: ys) \rightarrow \mathit{False}\} : list_1 \mathbin{<\!@\!>} t_2 \rightarrowtriangle bool_3}$$

$$\dfrac{\dfrac{\Gamma[xs \mapsto list_1 \mathbin{<\!@\!>} t_2](xs) = list_1 \mathbin{<\!@\!>} t_2}{\Gamma[xs \mapsto list_1 \mathbin{<\!@\!>} t_2] \vdash xs : list_1 \mathbin{<\!@\!>} t_2} \qquad \dfrac{\mathit{fresh}\ (1,2)}{\Gamma[xs \mapsto list_1 \mathbin{<\!@\!>} t_2] \vdash [\,] : list_1 \mathbin{<\!@\!>} t_2}}{\text{left subtree}}$$

$$\dfrac{\dfrac{\mathit{fresh}\ (3)}{\Gamma[xs \mapsto list_1 \mathbin{<\!@\!>} t_2\ ] \vdash \mathit{True} : bool_3} \qquad \dfrac{\mathit{fresh}\ (3)}{\Gamma[xs \mapsto list_1 \mathbin{<\!@\!>} t_2\ , y \mapsto t_2, ys \mapsto list_1 \mathbin{<\!@\!>} t_2\ ] \vdash \mathit{False} : bool_3}}{\text{right subtree}}$$

3.4.2 Unification and substitutions

As in Algorithm $\mathcal{W}$, we use Robinson's unification algorithm to generate substitutions during inferencing. For completeness, Figure 3.5 shows the grammar for substitutions, while Figure 3.6 shows the unification rules for contracts.

$$
\begin{array}{lll}
\theta ::= & \mathit{Id} & \text{-- Identity substitution} \\
\quad\mid & \theta_1 \circ \theta_2 & \text{-- Substitution composition} \\
\quad\mid & [c_1 \mapsto c_2] & \text{-- Substitution for } c_1 \text{ with a contract } c_2
\end{array}
$$

Figure 3.5: Grammar for substitutions

```
$\mathcal{U} : (\mathit{Contract}, \mathit{Contract}) \to \mathit{Substitution}$
$\mathcal{U}\ (c,\ c)\ \ = \mathit{Id}$
$\mathcal{U}\ (c_1, c_2) = [c_1 \mapsto c_2]\ (\mathit{iff}\, c_1 \notin fc(c_2) \wedge [\![c_2]\!] \subseteq [\![c_1]\!])$
$\mathcal{U}\ (c_1, c_2) = [c_2 \mapsto c_1]\ (\mathit{iff}\, c_2 \notin fc(c_1) \wedge [\![c_1]\!] \subseteq [\![c_2]\!])$
$\mathcal{U}\ (c_1 \rightarrowtriangle c_2, c_3 \rightarrowtriangle c_4) =$
    $\mathbf{let}\ \theta_1 = \mathcal{U}\ (c_1, c_3)$
          $\theta_2 = \mathcal{U}\ (\theta_1\ c_2, \theta_1\ c_4)$
    $\mathbf{in}\ \ \theta_2 \circ \theta_1$
$\mathcal{U}\ (c_1 <@> c_2)\ (c_3 <@> c_4) =$
    $\mathbf{let}\ \theta_1 = \mathcal{U}\ (c_1, c_3)$
          $\theta_2 = \mathcal{U}\ (\theta_1\ c_2, \theta_1\ c_4)$
    $\mathbf{in}\ \ \theta_2 \circ \theta_1$
$\mathcal{U}\ (c_1 <@@> (c_2, c_3), c_4 <@@> (c_5, c_6)) =$
    $\mathbf{let}\ \theta_1 = \mathcal{U}\ (c_1, c_4)$
          $\theta_2 = \mathcal{U}\ (\theta_1\ c_2, \theta_1\ c_5)$
          $\theta_3 = \mathcal{U}\ (\theta_2\ \theta_1\ c_3, \theta_2\ \theta_1\ c_6)$
    $\mathbf{in}\ \ \theta_3 \circ \theta_2 \circ \theta_1$
$\mathcal{U}\ (\_, \_)\ = \bot$
```

Figure 3.6: Unification rules for contracts

Unification for two identical contracts and for contract arrows is identical to the way one would unify types. We add two special cases for functors and bifunctors. These extra cases are similar to the case for contract arrows. First, we unify the outer contracts, after which we unify the inner contracts, apply substitutions, and finally return the composition of the new substitutions. More interesting are the two cases for unifying two different contracts. These cases are the same as in Algorithm $\mathcal{W}$, except for one extra condition. In order for c_1 to be unified with c_2, we require c_1 to be a subset of c_2, or the other way around. Using this condition, we can refine contracts upon unification. For example

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

would generate a substitution

$$[int \mapsto nat]$$

since $[\![nat]\!] \subseteq [\![int]\!]$.

We need to take care to generate the correct substitutions when unifying. In types, unification is straight-forward. Unifying $a \to b$ and $c \to d$ will correctly generate substitutions $([\,a \mapsto c] \circ [\,b \mapsto d])$, because type variables can be distinguished by name. With contracts, unification is harder, because instead of fresh type variables, we assign concrete contracts. Why this is harder is illustrated by the following example:

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

Unification in this example would proceed as follows. First, we $\mathcal{U}\ (true, int)$ giving a substitution $[true \mapsto int]$, which is then applied to both the second $true$ and nat, replacing the second $true$ with int. Next, we $\mathcal{U}\ (int, nat)$, which would give us a substitution $[int \mapsto nat]$. Applying these substitutions to the left-hand contract would give the following result:

$$([int \mapsto nat] \circ [true \mapsto int])\ (true \twoheadrightarrow true) = nat \twoheadrightarrow nat$$

In the situation where we see the $true \twoheadrightarrow true$ contract analogous to $a \to a$ in types, the unification result is actually the desired outcome. However, if we wanted the $true \twoheadrightarrow true$ contract to be analogous to $a \to b$ in types, we would want the following contract as a result of the unification:

$$int \twoheadrightarrow nat$$

For this to be possible, we need to be able to distinguish between the same concrete contracts, e.g., between the one $true$ contract and another $true$ contract. This is exactly what Definition 1 allows us to do by adding indices to contracts. If instead of $true \twoheadrightarrow true$, we have $true_1 \twoheadrightarrow true_2$, unification would proceed differently:

$$\mathcal{U}\ (true_1 \twoheadrightarrow true_2, int_3 \twoheadrightarrow nat_4) = [true_1 \mapsto int_3] \circ [true_2 \mapsto nat_4]$$

and applying the substitution would give us a different answer:

$$([true_1 \mapsto int_3] \circ [true_2 \mapsto nat_4])\ (true_1 \twoheadrightarrow true_2) = int_3 \twoheadrightarrow nat_4$$

3.4.3 Algorithm $\mathcal{CW}$

Combining the inference rules and the unification algorithm described in the previous subsection, we can now define an inferencing algorithm which when given a λ_c expression, infers a contract of that expression. Since the algorithm is heavily based on Algorithm $\mathcal{W}$, we call it Algorithm $\mathcal{CW}$. We present it in Figure 3.7, but we omit the case blocks and binary operations, because they do not add anything new.

Our Algorithm $\mathcal{CW}$ differs from Algorithm $\mathcal{W}$ in several ways. Firstly, $\mathcal{CW}$ explicitly implements inference for data types and holes. Secondly, it explicitly implements **let** as a recursive **let**, whereas the implementation for **let** by Damas and Milner is non-recursive. Their argument for omitting a recursive **let** was that adding it is simple and they wanted to keep their language in the paper minimal. We choose to explicitly model a recursive **let**, because Haskell's **let** is also recursive and we want the step to implementing contract inference for Haskell to be as small as possible. Lastly, rather than generating fresh variables, we generate fresh indexed *true* contracts, which can be refined by unification.

```
$\mathcal{CW} : \mathit{Environment} \to \mathit{Expression} \to (\mathit{Substitution}, \mathit{Contract})$
$\mathcal{CW}\ \Gamma\ x$                                 $=$ if $x \in \mathit{dom}\ (\Gamma)$
                                          then $(\mathit{Id}, \mathit{inst}\ \Gamma\ (x))$
                                          else $\bot$
$\mathcal{CW}\ \Gamma\ (\lambda x \to e)$                  $=$ let $i$ be fresh
                                          $(\theta, c) = \mathcal{CW}\ (\Gamma\ [x \mapsto \mathit{true}_i])\ e$
                                       in $(\theta, \theta\ \mathit{true}_i \rightarrowtriangle c)$
$\mathcal{CW}\ \Gamma\ (e_1\ e_2)$                         $=$ let $i$ be fresh
                                          $(\theta_1, c_1 \rightarrowtriangle c) = \mathcal{CW}\ \Gamma\ e_1$
                                          $(\theta_2, c_2)\qquad = \mathcal{CW}\ (\theta_1\ \Gamma)\ e_2$
                                          $\theta_3\qquad\qquad = \mathcal{U}\ (\theta_2\ c_1 \rightarrowtriangle c, c_2 \rightarrowtriangle \mathit{true}_i)$
                                       in $(\theta_3 \circ \theta_2 \circ \theta_1, \theta_3\ \mathit{true}_i)$
$\mathcal{CW}\ \Gamma\ (\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2)$ $=$ let $i$ be fresh
                                          $(\theta_1, c_1) = \mathcal{CW}\ (\Gamma\ [x \mapsto \mathit{true}_i])\ e_1$
                                          $\theta_2\qquad = \mathcal{U}\ (\theta_1\ \mathit{true}_i, c_1)$
                                          $(\theta_3, c)\ = \mathcal{CW}\ (\theta_2 \circ \theta_1\ \Gamma\ [x \mapsto \mathit{gen}\ (\theta_2 \circ \theta_1\ \Gamma)\ \theta_2\ c_1])\ e_2$
                                       in $(\theta_3 \circ \theta_2 \circ \theta_1, c)$
$\mathcal{CW}\ \Gamma\ [\,]$                              $=$ let $i, j$ be fresh
                                       in $(\mathit{Id}, \mathit{list}_i \mathbin{<@>} \mathit{true}_j)$
$\mathcal{CW}\ \Gamma\ (x :: \mathit{xs})$                  $=$ let $(\theta_1, c)\qquad\qquad = \mathcal{CW}\ \Gamma\ x$
                                          $(\theta_2, \mathit{list}_i \mathbin{<@>} c) = \mathcal{CW}\ \Gamma\ \mathit{xs}$
                                       in $(\theta_2 \circ \theta_1, \mathit{list}_i \mathbin{<@>} c)$
$\mathcal{CW}\ \Gamma\ \mathit{Nothing}$                     $=$ let $i, j$ be fresh
                                       in $(\mathit{Id}, \mathit{maybe}_i \mathbin{<@>} \mathit{true}_j)$
$\mathcal{CW}\ \Gamma\ (\mathit{Just}\ x)$                   $=$ let $i$ be fresh
                                          $(\theta, c) = \mathcal{CW}\ \Gamma\ x$
                                       in $(\theta, \mathit{maybe}_i \mathbin{<@>} c)$
$\mathcal{CW}\ \Gamma\ (x, y)$                             $=$ let $i$ be fresh
                                          $(\theta_1, c_1) = \mathcal{CW}\ \Gamma\ x$
                                          $(\theta_2, c_2) = \mathcal{CW}\ \Gamma\ y$
                                       in $(\theta_2 \circ \theta_1, \mathit{pair}_i \mathbin{<@@>} (c_1, c_2))$
$\mathcal{CW}\ \Gamma\ (\mathit{Left}\ x)$                   $=$ let $i, j$ be fresh
                                          $(\theta, c) = \mathcal{CW}\ \Gamma\ x$
                                       in $(\theta, \mathit{either}_i \mathbin{<@@>} (c, \mathit{true}_j))$
$\mathcal{CW}\ \Gamma\ (\mathit{Right}\ x)$                  $=$ let $i, j$ be fresh
                                          $(\theta, c) = \mathcal{CW}\ \Gamma\ x$
                                       in $(\theta, \mathit{either}_i \mathbin{<@@>} (\mathit{true}_j, c))$
$\mathcal{CW}\ \Gamma\ ?$                                 $=$ let $i$ be fresh
                                       in $(\mathit{Id}, \mathit{true}_i)$
```

Figure 3.7: Algorithm $\mathcal{CW}$

We show that Algorithm $\mathcal{CW}$ is sound with respect to contracting rules in Figure 3.4 in Proposition 5.

Proposition 5 (Soundness of inference)**.** *If* $\mathcal{CW}\ \Gamma\ e = (\theta, c)$*, then* $\Gamma \vdash e : c$.

Proof. Proof by induction on e. The full proof is given in Appendix A.1. $\square$

Asserting inferred contracts

Algorithm $\mathcal{CW}$ infers contracts in our intermediate contract language. Before these contracts can be asserted, they need to be translated to executable program code for a particular contract library. Since our work is based on the work by Hinze et al. and our contract language is inspired by their notation, the translation between the contract language and the contract library types is easy. Contracts for *true*, $false$, *int*, *bool*, *list*, *either*, etc. are translated into an executable form, maintaining their semantics. The contract arrow, functor and bifunctor contracts have a direct translation to the contract library. Contract schemes are instantiated first, before being translated into executable code.

Once inferred contracts have been translated into an executable form, we can show in Proposition 6 that if Algorithm $\mathcal{CW}$ infers a contract for an expression e, the inferred contract will never fail assertion for that expression. In other words, applying the *assert* function to the inferred contract will give the identity function for expression e.

Proposition 6 (Asserting inferred contract is identity)**.** *If* $\mathcal{CW}\ \Gamma\ e = (\theta, c)$*, then* $assert\ c\ e = e$.

Proof. Proof by induction on e. The full proof is given in Appendix A.2. Conversion to an executable contract is left implicit. $\square$

Not only is asserting the inferred contract the identity, the inferred contract is also the most specific contract. I.e., the semantics of the inferred contract is a subset of any other contract that can be described in the contract system. We formulate this in Conjecture 1. Intuitively, this seems true, because $\mathcal{CW}$ will infer contracts specific to certain types. E.g., it will infer a functor contract with a *list* outer contract for lists. However, this is not the only valid contract for a list in the contract system. We can also replace the functor contract with a *true* contract. While this is less specific, it will still be a valid contract for a list, since $[\![list \lessdot@\gtrdot true]\!] \subseteq [\![true]\!]$. A formal proof for this conjecture is left for future work.

Conjecture 1 (An inferred contract is the most specific)**.** *If* $\mathcal{CW}\ \Gamma\ e = (\theta, c)$ *and* $\Gamma \vdash e : c$*, then for all* $\Gamma \vdash e : c'$*,* $[\![c]\!] \subseteq [\![c']\!]$.

3.5 Dependent contracts

So far, we have only dealt with contracts in the form of

$$c_1 \rightarrowtriangle c_2$$

However, such contracts cannot capture all properties of a function. Contracts that capture the properties of a function completely commonly rely on a function's *input*, i.e., they are *dependent contracts*. Consider again the *sort* function. We can define a contract

$$(list \lessdot\!@\!\gtrdot true) \multimap (ord \lessdot\!@\!\gtrdot true)$$

that can correctly identify problems when *sort*, or one of the functions it uses, does not return an ordered list. However, if we implement *sort* as

$$sort\ xs = [\,]$$

it trivially satisfies its contract, since an empty list is always sorted, and the contract will never fail. Still, it is clear that this function is not a sorting function, as it just ignores the input.

This problem is due to the fact that we have not made our contract specific enough. Sorting a list does not only mean delivering a sorted list, it also means that the resulting list is a permutation of the input list. In other words, the contract for the function's result *depends* on the function's input. We can capture the correct contract for *sort* in a dependent contract

$$(xs : list \lessdot\!@\!\gtrdot true) \xmapsto{d} (sorted\ xs)$$

where *sorted* is a contract that checks whether the output is sorted and is a permutation of the input list *xs*. Recapping from Section 2.2.1, the dependent contract arrow ($\xmapsto{d}$) is defined as

$$(\xmapsto{d}) = Function$$

where *Function* is a constructor from the *Contract* GADT, which allows the function contract's argument to be used in its result. It is the same constructor used for the non-dependent contract arrows ($\multimap$), with the only difference that ($\multimap$) applies *const* to the second argument. One might be tempted to simply replace the non-dependent contract arrow with a dependent contract arrow in the inferencing algorithm. As we will see shortly, this is not enough to successfully infer dependent contracts. For example, suppose *sort* was implemented in terms of *foldr*:

$$foldr : (a \to b \to b) \to b \to [\,a\,] \to b$$

Now imagine that the dependent contract for *sort* was naively unified with the contract of *foldr*. Doing so would produce the following contract for *foldr*, where y, ys and b are freshly generated variable names:

$$((y : true) \xmapsto{d} (ys : sorted\ xs) \xmapsto{d} (sorted\ xs)) \xmapsto{d} (b : sorted\ xs)$$
$$\xmapsto{d} (xs : list \lessdot\!@\!\gtrdot true) \xmapsto{d} (sorted\ xs)$$

Several problems plague this contract. Firstly, xs is bound only in the second-last argument, but it is already referred to before it comes into scope. Secondly, the contract for ys also requires a list xs, but since there simply is no list argument defined before b, this is impossible. Lastly, the contract for the result of the function argument should be ($sorted\ (y :: ys)$), instead of ($sorted\ xs$), because *sort*'s *insert* function produces a sorted list by inserting the element y into the list ys.

We can solve the first and second problem by only assigning contracts to function *results*. However, it is unclear how to solve the last problem. The following contract is the desired dependent contract for *foldr* in the context of the *sort* function:

$$((y : true_1) \overset{d}{\longmapsto} (ys : true_2) \overset{d}{\longmapsto} (sorted\ (y :: ys))) \overset{d}{\longmapsto} (b : true_2)$$
$$\overset{d}{\longmapsto} (xs : true_3) \overset{d}{\longmapsto} (sorted\ xs)$$

If at all possible, inferring this contract from code is a hard problem which we will not attempt to solve in this thesis. For this reason, we do not attempt to infer dependent contracts at all.

3.5.1 Eliminating dependent contracts

Working around the problem of inferring dependent contracts, rather than solving it, is also possible in certain cases. Instead of trying to infer dependent contracts, we get rid of them altogether, eliminating the problems described above. We do so by inlining a QuickCheck counter-example in a function's contract. The intuition behind this approach is that since QuickCheck has found a counter-example, contract assertion fails for the same input. For this to work, it is essential that the QuickCheck properties and the contracts are exactly the same.

To understand why using a QuickCheck counter-example allows us to eliminate dependent contracts, we need to understand how QuickCheck produces a counter-example. When starting a QuickCheck test, QuickCheck will generate random values and apply the property under test to them. When QuickCheck has generated a value that violates the property under test, it will try to *shrink* it, i.e., it will try to make the counter-example structurally as small as possible, ensuring that it keeps violating the property under test. It does this, because the values it finds are random values, not necessarily minimal counter-examples. For example, suppose QuickCheck has randomly generated the list

$$[0, 1, 2]$$

and has determined that this value causes the property under test to fail. Before returning this value to the user, QuickCheck will try to systematically reduce the number of elements in the list, until it finds the smallest list that causes the property to fail. In this specific example, it will first try all of the values from the following list[1]:

$$[[\,], [1,2], [0,2], [0,1], [0,0,2], [0,1,0], [0,1,1]]$$

It will then try to shrink all of these lists. For example, when it tries to shrink the list $[1,2]$, it will generate the list

$$[[\,], [2], [1], [0,2], [1,0], [1,1]]$$

From which each element is tested again. Eventually QuickCheck will return one of the smallest counter-examples it can find. Indeed, in QuickCheck's current implementation there is no guarantee that it is a minimal counter-example. We will discuss this problem in Section 4.1. In the mean time, we will consider the counter-example returned by QuickCheck to be minimal in order to demonstrate how we work around dependent contracts.

When we have a minimal counter-example, e.g. $[0,1]$, we can assume that the property under test succeeds for the smaller list $[1]$. We can use this knowledge when embedding contract assertions in our program. Suppose we have a function

```
$f : [Int] \to [Int]$
$f\ xs = assert\ c\ g\ xs$
  **where** $g\ [\,]$        $= [\,]$
        $g\ (y :: ys)$ $= (y + 1 :: assert\ c\ g\ ys)$
        $c$             $= (xs' : true) \overset{d}{\longmapsto} (silly\ xs')$
```

where *silly* is some contract that fails for $xs' = [0,1]$ (and possibly larger lists), but not for $xs' = [1]$ and $xs' = [\,]$. Now we want to eliminate the dependent contract. Since QuickCheck gives us $[0,1]$ as counter-example for this function, we inline it in the place of xs' in the *silly* contract, and replace the dependent contract arrow with a plain contract arrow, resulting in the following code:

```
$f : [Int] \to [Int]$
$f\ xs = assert\ c\ g\ xs$
  **where** $g\ [\,]$        $= [\,]$
        $g\ (y :: ys)$ $= (y + 1 :: g\ ys)$
        $c$             $= true \twoheadrightarrow (silly\ [0,1])$
```

Note that we do not assert the contract in recursive applications of g. Since we know that QuickCheck's counter-example is minimal, we do not have to, because having a minimal counter-example implies that the contract will succeed for smaller lists. Now when we apply f to $[0,1]$, we can expect the *silly* contract to fail and blame the right location. While this particular example is

[1]The list is generated by QuickCheck's *shrink* function. It does not necessarily contain all permutations of the original list

rather contrived, experimentation with the *sort* function has indicated that this approach works for bigger examples as well.

So far, we only have initial experimental results for eliminating dependent contracts. Section 4.1 will explore some of the problems that affect this idea.

Chapter 4

Discussion and future work

We have seen how contracts can be inferred from functional programs. We think that the ideas presented in this thesis can be extended to a full-fledged programming language, such as Haskell, and lifted outside the context of the Ask-Elle tutor. Still, there are several aspects of contract inferencing we have not fully explored. In this section we will look at aspects of contract inferencing that we have not fully explored and the shortcomings of the current implementation of our system.

4.1 Exploring dependent contracts

Section 3.5 discussed dependent contracts, the problems with dependent contracts and contract inference, and how we can work around the need for them using a QuickCheck counter-example. While the results in that section look promising, only few experiments have been performed. Further experimentation will be required to verify that the idea presented in that section is valid in general, and that it can be applied to most or all contracts and programs.

One of the problems identified in Section 3.5 was that QuickCheck currently does not necessarily give a minimal counter-example. This is due to the current implementation of QuickCheck's *Arbitrary* class instances, the type class which generates the random values. This particular explanation was given by Nick Smallbone on the QuickCheck mailing list. Suppose we have some function f and a property p for f. Now suppose that p fails for the list $[0,0,0]$, succeeds for $[0,0]$, but fails again for $[0]$. QuickCheck will always shrink $[0,0,0]$ to $[0,0]$ first, before shrinking it to $[0]$. In the example, QuickCheck will never try shrinking to $[0]$, because $[0,0]$ has already succeeded. To make this more concrete, f and p could be implemented as follows:

```
f [_]       = False
f [_, _, _] = False
f _         = True
```

$p\ xs = f\ xs$

Outside the context of our work, QuickCheck's current shrinking behaviour is desired, because it reduces the number of shrink operations, speeding up testing. However, since we are interested in minimal examples, we want QuickCheck to also shrink $[0, 0]$, so we can find the even smaller counter-example $[0]$. Different shrinking behaviour can be obtained by re-implementing the instances for the *Arbitrary*.

Another problem with relying on QuickCheck for shrinking a counter-example is that it is not always clear what it means for a counter-example to be the smallest. For lists, it's clear: the empty list is the smallest lists one can have. If natural numbers in Haskell were defined as

data *Nat* = *Succ Nat* | *Zero*

then it would be clear that *Zero* is the smallest natural number. However, Haskell only has *Int* and *Integer* to represent whole numbers. QuickCheck will shrink integers towards 0. For example, suppose it finds (-7) as counter-example, it will produce the following shrink list:

$[7, 0, -4, -6]$

while another perfectly reasonable shrink list would have been

$[-11, -13, -17, -19]$

There is no guarantee that either of these lists contain a "smallest" number such that the function under test needs to make a minimum number of recursive calls before it terminates. This is illustrated by the following function f, which generates an infinite list of integers:

$f : Int \rightarrow [Int]$
$f\ n = n :: f\ (n + 1)$

Even though *Int* has a definition for *maxBound*, simply increasing the number of n will eventually cause the number's sign bit to flip, appending the *minBound* value for *Int* to the list, after which the function will continue until it reaches the *maxBound* for *Int* again. f will never terminate. From this we can conclude that it is hard to define what it means for a number to be the smallest, and our approach for eliminating dependent contracts in its current form will probably not work for *Int*. A similar argument holds for *Integer*, because it is not bounded. How we can eliminate dependent contracts when integers (and possibly other types) are used remains an open question.

4.2 Embedding contract inference in Ask-Elle

Our original motivation for this work was the Ask-Elle Haskell tutor. By combining QuickCheck and contract inferencing, we wanted to locate where a program failed its contracts when the Ask-Elle strategies could no longer do so. Integrating contract inference into Ask-Elle still remains to be done.

In the context of the tutor, we want the student to focus only on learning. The student should not be required to do any extra work to make use of contract inferencing. As such, we want the tutor to automatically infer contracts and augment the student's program with the inferred contracts, so they may be asserted by applying the student's function to a QuickCheck counter-example. Initial experimentation has shown that it is possible to do so, although it remains to be seen if it is possible to define transformations that achieve this in general.

As we saw in Section 3.5, we require QuickCheck counter-examples in order to eliminate dependent contracts. However, out of the box, QuickCheck can only *display* the counter example it finds, and not return it for further use. Koen Claessen, one of the original authors of QuickCheck, proposed a workaround on the QuickCheck mailing list. The workaround involves wrapping QuickCheck's *quickCheck* function–the function that starts testing the specified property–by a custom function that uses an *IORef* that stores the counter-examples QuickCheck produces while testing.

```
quickCheckArg : (Arbitrary a, Testable prop) ⇒ (a → prop) → IO (Maybe a)
quickCheckArg p = do
  ref ← newIORef Nothing
  quickCheck (λx → whenFail (writeIORef ref (Just x)) (p x))
  readIORef ref
```

We can then read the counter-example values from the *Maybe* value.

Lastly, when integrating contract inference with the tutor, we will need to be able to infer more interesting contracts than those that never fail assertion. Teachers need to be able to specify a contract for the exercise, and the contract inferencing algorithm needs to be able to use this specific contract to infer contracts for the rest of the program. In essence, this is not very different from the way an explicit type annotation is persisted through a program in type inference. Our implementation already largely supports using explicit contracts during inference, although we still need to finish the implementation and integrate this with the tutor.

4.3 Constant expression contracts

Contracts inferred using Algorithm $\mathcal{CW}$ are not very useful if they always succeed. However, in some cases, we may be able to infer more specific contracts from the code, making it more likely that we will find a situation where an assertion fails. Consider the following (contrived) example:

```
silly : a → Int
silly x = 1
```

The resulting tuple will always have an integer 1 for its left element. A perfectly valid and reasonable contract for *silly* would therefore be:

```
true ⇾ equalsOne
```

Where *equalsOne* is an inferred contract that only succeeds for the constant value 1. We can easily infer this contract from the code. Such contracts could be called *constant expression contracts*, as they check for constant values. Unfortunately, naively generating these constant expression contracts will quickly lead to problematic contracts. Consider, for example, a constant expression contract for *silly2*:

$$\begin{array}{lll} \mathit{silly2}\ \mathit{xs} = \textbf{let} & f\ [\,] & = [\,] \\ & f\ (y :: \mathit{ys}) & = (\textbf{if}\ y \equiv 1\ \textbf{then}\ y+1\ \textbf{else}\ 0) :: f\ (\mathit{map}\ (+1)\ \mathit{ys}) \\ \qquad\qquad\quad \textbf{in} & f\ (\mathit{map}\ (\lambda y \rightarrow 1)\ \mathit{xs}) & \end{array}$$

Using constant expression inferencing, we can infer the contract

$$(\mathit{true} \rightarrowtriangle \mathit{equalsOne}) \rightarrowtriangle (\mathit{list}_1 \mathbin{<\!@\!>} \mathit{true}) \rightarrowtriangle (\mathit{list}_2 \mathbin{<\!@\!>} \mathit{equalsOne})$$

for the first application of *map* in the **in** branch of the **let** block, which will be unified with the contract for f. If we would infer a contract for f alone, it would get contract

$$(\mathit{list}_1 \mathbin{<\!@\!>} \mathit{int}_2) \rightarrowtriangle (\mathit{list}_3 \mathbin{<\!@\!>} \mathit{int}_2)$$

due to its type

$$f : (\mathit{Eq}\ a, \mathit{Num}\ a) \Rightarrow [\,a\,] \rightarrow [\,a\,]$$

After unification with *map*'s contract, f gets the following contract:

$$(\mathit{list}_1 \mathbin{<\!@\!>} \mathit{equalsOne}) \rightarrowtriangle (\mathit{list}_3 \mathbin{<\!@\!>} \mathit{equalsOne})$$

This contract is clearly not correct, because the list f produces never contains a 1 as element. Inferring constant expression contracts in this form requires program analysis to ensure that values remain constant. However, this is outside the scope of this thesis.

One possible solution to this problem would be to not allow constant expression contracts to be unified. Instead, upon unification, a superset contract could be used in the substitution. For example, when unifying

$$\mathit{list}_1 \mathbin{<\!@\!>} \mathit{equalsOne}$$

and

$$\mathit{list}_1 \mathbin{<\!@\!>} \mathit{int}_2$$

we could opt to maintain the int_2 contract, rather that generating a $[\mathit{int}_2 \mapsto \mathit{equalsOne}]$ substitution. Clearly, this problem is under-explored and more work is required to before useful constant expression contracts can be inferred.

Appendix A

Proofs

A.1 Soundness of Algorithm $\mathcal{CW}$

We want to show that if $\mathcal{CW}\ \Gamma\ e = (\theta, c)$, then $\Gamma \vdash e : c$. For variables, application, abstraction, and **let** bindings we refer to Damas and Milner's original proof, as contract inference and type inference are the same for these rules. We also forego explicitly proving soundness for $[\,]$, *Nothing*, and ?, since these proofs are trivial. We also omit the proof for case blocks and binary operations, since they are not represented in presentation of $\mathcal{CW}$. Lastly, we also omit the proof for *Right*, as it is very similar to the proof for *Left*. For the remaining expressions, we proceed by induction on e. We assume fresh contract variables to be tautologies.

Case $(x :: xs)$

To show: if $\mathcal{CW}\ \Gamma\ (x :: xs) = (\theta_2 \circ \theta_1, \mathit{list} <\!@\!> c)$, then $\Gamma \vdash (x :: xs) : \mathit{list} <\!@\!> c$

IH1 If $\mathcal{CW}\ \Gamma\ x = (\theta_1, c)$, then $\Gamma \vdash x : c$

IH2 If $\mathcal{CW}\ \Gamma\ xs = (\theta_2, \mathit{list} <\!@\!> c)$, then $\Gamma \vdash xs : \mathit{list} <\!@\!> c$

Proof:

$$\begin{array}{ll}
 & \Gamma \vdash (x :: xs) : \mathit{list} <\!@\!> c \\
\Leftarrow & \quad \{ \text{ Rule C-}\textsc{Cons} \} \\
 & \Gamma \vdash x : c, \Gamma \vdash xs : \mathit{list} <\!@\!> c \\
\Leftarrow & \quad \{ \text{ Case } x \text{, apply IH1 } \} \\
 & \mathcal{CW}\ \Gamma\ x = (\theta_1, c) \\
\Leftarrow & \quad \{ \text{ Case } xs \text{, apply IH2 } \} \\
 & \mathcal{CW}\ \Gamma\ xs = (\theta_2, \mathit{list} <\!@\!> c)
\end{array}$$

from which follows that if $\mathcal{CW}\ \Gamma\ (x :: xs) = (\theta_2 \circ \theta_1, \mathit{list} <\!@\!> c)$, then $\Gamma \vdash (x :: xs) : \mathit{list} <\!@\!> c$.

Case *Just x*

To show: if $\mathcal{CW}\ \Gamma\ (\mathit{Just}\ x) = (\theta, \mathit{maybe} <\!\!@\!\!> c)$, then $\Gamma \vdash \mathit{Just}\ x : \mathit{maybe} <\!\!@\!\!> c$.

IH If $\mathcal{CW}\ \Gamma\ x = (\theta, c)$, then $\Gamma \vdash x : c$

Proof:

$$\begin{array}{ll} & \Gamma \vdash \mathit{Just}\ x : \mathit{maybe} <\!\!@\!\!> c \\ \Leftarrow & \quad \{ \text{ Rule C-}\textsc{Just} \text{ } \} \\ & \Gamma \vdash x : c \\ \Leftarrow & \quad \{ \text{ Apply IH } \} \\ & \mathcal{CW}\ \Gamma\ x = (\theta, c) \end{array}$$

from which follows that if $\mathcal{CW}\ \Gamma\ (\mathit{Just}\ x) = (\theta, \mathit{maybe} <\!\!@\!\!> c)$, then $\Gamma \vdash \mathit{Just}\ x : \mathit{maybe} <\!\!@\!\!> c$.

Case (x, y)

To show: if $\mathcal{CW}\ \Gamma\ (x, y) = (\theta_2 \circ \theta_1, \mathit{pair} <\!\!@@\!\!> (c_1, c_2))$, then $\Gamma \vdash (x, y) : \mathit{pair} <\!\!@@\!\!> (c_1, c_2)$.

IH1 If $\mathcal{CW}\ \Gamma\ x = (\theta_1, c_1)$, then $\Gamma \vdash x : c_1$

IH2 If $\mathcal{CW}\ \Gamma\ y = (\theta_2, c_2)$, then $\Gamma \vdash y : c_2$

Proof:

$$\begin{array}{ll} & \Gamma \vdash (x, y) : \mathit{pair} <\!\!@@\!\!> (c_1, c_2) \\ \Leftarrow & \quad \{ \text{ Rule C-}\textsc{Pair} \text{ } \} \\ & \Gamma \vdash x : c_1, \Gamma \vdash y : c_2 \\ \Leftarrow & \quad \{ \text{ Case } x\text{, apply IH1 } \} \\ & \mathcal{CW}\ \Gamma\ x = (\theta_1, c_1) \\ \Leftarrow & \quad \{ \text{ Case } y\text{, apply IH2 } \} \\ & \mathcal{CW}\ \Gamma\ y = (\theta_2, c_2) \end{array}$$

from which follows that if $\mathcal{CW}\ \Gamma\ (x, y) = (\theta_2 \circ \theta_1, \mathit{pair} <\!\!@@\!\!> (c_1, c_2))$, then $\Gamma \vdash (x, y) : \mathit{pair} <\!\!@@\!\!> (c_1, c_2)$.

Case *Left x*

To show: if $\mathcal{CW}\ \Gamma\ (\mathit{Left}\ x) = (\theta, \mathit{either} <\!\!@@\!\!> (c, \mathit{true}_i))$, then $\Gamma \vdash \mathit{Left}\ x : \mathit{either} <\!\!@@\!\!> (c, \mathit{true}_i)$.

IH If $\mathcal{CW}\ \Gamma\ x = (\theta, c)$, then $\Gamma \vdash x : c$

Proof:

$$\begin{array}{ll}
& \Gamma \vdash \mathit{Left}\ x : \mathit{either} <@@> (c, \mathit{true}_i) \\
\Leftarrow & \{ \text{ Rule C-L}\textsc{eft} \ \} \\
& \Gamma \vdash x : c, \mathit{fresh}\ (i) \\
\Leftarrow & \{ \text{ Apply IH } \} \\
& \mathcal{CW}\ \Gamma\ x = (\theta, c)
\end{array}$$

from which follows that if $\mathcal{CW}\ \Gamma\ (\mathit{Left}\ x) = (\theta, \mathit{either} <@@> (c, \mathit{true}_i))$, then $\Gamma \vdash \mathit{Left}\ x : \mathit{either} <@@> (c, \mathit{true}_i)$.

A.2 Asserting inferred contract is identity

We want to show that if $\mathcal{CW}\ \Gamma\ e = (\theta, c)$, then $\mathit{assert}\ c\ e = e$. We forego providing explicit proofs for variables, $[\,]$, *Nothing*, and ?, since these proofs are trivial. We also omit the proof for case blocks and binary operations, since they are not represented in presentation of $\mathcal{CW}$. Lastly, we also omit the proof for *Right*, as it is very similar to the proof for *Left*. For the remaining expressions, we proceed by induction on e.

For lambda abstraction and function application we failed to give a full proof, due to the important role unification and substitutions play in these parts of the algorithm. It was not clear how to incorporate these in the proofs. Instead of completing the proof, we opted to give an informal reasoning as to why we think the proof can be completed.

Case $\lambda x \to e$

If $\mathcal{CW}\ \Gamma\ (\lambda x \to e) = (\theta, \theta\ \mathit{true}_i \twoheadrightarrow c)$, then $\mathit{assert}\ (\theta\ \mathit{true}_i \twoheadrightarrow c)\ (\lambda x \to e) = (\lambda x \to e)$

IH If $\mathcal{CW}\ (\Gamma\ [x \mapsto \mathit{true}_i])\ e = (\theta, c_2)$, then $\mathit{assert}\ c_2\ e = e$

Proof:

$$\begin{array}{ll}
& \mathit{assert}\ ((\theta\ \mathit{true}_i) \twoheadrightarrow c_2)\ (\lambda x \to e) \\
\equiv & \{ \text{ Definition of } \twoheadrightarrow \} \\
& \mathit{assert}\ (\mathit{Function}\ (\theta\ \mathit{true}_i)\ (\mathit{const}\ c_2))\ (\lambda x \to e) \\
\equiv & \{ \text{ Definition of } \mathit{assert} \ \} \\
& \lambda x' \to (\mathit{assert}\ (\mathit{const}\ c_2\ x') \circ \lambda x \to e)\ x' \circ \mathit{assert}\ (\theta\ \mathit{true}_i) \\
\equiv & \{ \text{ Definition of } \mathit{const} \ \} \\
& \lambda x' \to (\mathit{assert}\ c_2 \circ \lambda x \to e)\ x' \circ \mathit{assert}\ (\theta\ \mathit{true}_i) \\
\equiv & \{ \text{ Apply } x' \ \} \\
& \lambda x' \to (\mathit{assert}\ c_2\ (e\ [x\ /\ x'])) \circ \mathit{assert}\ (\theta\ \mathit{true}_i) \\
\equiv & \{ \text{ Missing steps } \} \\
& \lambda x' \to \mathit{assert}\ c_2\ (e\ [x\ /\ x'])
\end{array}$$

$$\begin{aligned}
\equiv\quad & \{\text{ Apply IH }\} \\
& \lambda x' \rightarrow e\ [x \,/\, x']
\end{aligned}$$

Part of this proof is missing, because it could not be proved in time. Instead of a formal proof, we give an informal reasoning for why we think it is possible to give a proof for this case. We can reason that a valid contract will be inferred by $\mathcal{CW}$ for x somewhere in expression e. A substitution will be returned which will substitute the inferred contract for the fresh contract $true_i$. By this Proposition, we then know that asserting this inferred contract yields the identity of the value for which we assert this contract. This allows us to eliminate the right-most assertion:

$$\begin{aligned}
& \lambda x' \rightarrow (\mathit{assert}\ c_2\ (e\ [x \,/\, x'])) \circ \mathit{assert}\ (\theta\ \mathit{true}_i) \\
\equiv\quad & \{\ \mathit{assert} \text{ is identity }\} \\
& \lambda x' \rightarrow \mathit{assert}\ c_2\ (e\ [x \,/\, x'])
\end{aligned}$$

With all occurrences of x replaced by x'

$$\lambda x' \rightarrow \mathit{assert}\ c_2\ (e\ [x \,/\, x'])$$

is equivalent to

$$\lambda x \rightarrow \mathit{assert}\ c_2\ e$$

up to α-conversion, so we can apply our induction hypothesis, completing our proof.

Case $e_1\ e_2$

If $\mathcal{CW}\ \Gamma\ (e_1\ e_2) = (\theta_3 \circ \theta_2 \circ \theta_1, \theta_3\ \mathit{true}_i)$, then $\mathit{assert}\ (\theta_3\ \mathit{true}_i)\ (e_1\ e_2) = (e_1\ e_2)$

IH1 If $\mathcal{CW}\ \Gamma\ e_1 = (\theta_1, c_1 \twoheadrightarrow c)$, then $\mathit{assert}\ (c_1 \twoheadrightarrow c)\ e_1 = e_1$

IH2 If $\mathcal{CW}\ (\theta_1\ \Gamma)\ e_2 = (\theta_2, c_2)$, then $\mathit{assert}\ c_2\ e_2 = e_2$

Proof:

$$\begin{aligned}
& \mathit{assert}\ (\theta_3\ \mathit{true}_i)\ (e_1\ e_2) \\
\equiv\quad & \{\text{ Definition of } \theta_3 \ \} \\
& \mathit{assert}\ (\mathcal{U}\ (\theta_2\ c_1 \twoheadrightarrow c, c_2 \twoheadrightarrow \mathit{true}_i)\ \mathit{true}_i)\ (e_1\ e_2) \\
\equiv\quad & \{\text{ Definition of } \mathcal{U} \ \} \\
& \mathit{assert}\ ((\mathcal{U}\ (\theta_2\ c, \mathcal{U}\ (\theta_2\ c_1, c_2)\ \mathit{true}_i) \circ \mathcal{U}\ (\theta_2\ c_1, c_2))\ \mathit{true}_i)\ (e_1\ e_2) \\
\equiv\quad & \{\text{ Missing steps }\} \\
& \ldots \\
\equiv\quad & \{\text{ Case } e_1 \ \} \\
& \mathit{assert}\ (c_1 \twoheadrightarrow c)\ e_1
\end{aligned}$$

$\equiv$ { Apply IH1 }
  e_1
$\equiv$ { Case e_2 }
  $\mathit{assert}\ c_2\ e_2$
$\equiv$ { Apply IH2 }
  e_2

Again, part of the proof is missing and is left as future work. We assume that this proof can also be given, because e_1 will be some function and e_2 some other expression. If we finish the proof for lambda abstraction, we can reason that we can also complete this proof for e_1. When we then also assume that this case can be proved, we can also reason that the case for e_2 can be proved.

Case let $x = e_1$ in e_2

If $\mathcal{CW}\ \Gamma\ (\textbf{let}\ x = e_1\ \textbf{in}\ e_2) = (\theta_3 \circ \theta_2 \circ \theta_1, c)$, then $\mathit{assert}\ c\ (\textbf{let}\ x = e_1\ \textbf{in}\ e_2) = (\textbf{let}\ x = e_1\ \textbf{in}\ e_2)$

IH1 If $\mathcal{CW}\ (\Gamma\ [x \mapsto \mathit{true}_i])\ e_1 = (\theta_1, c_1)$, then $\mathit{assert}\ c_1\ e_1 = e_1$

IH2 If $\mathcal{CW}\ (\theta_2\ \circ\ \theta_1\ \Gamma\ [x \mapsto \mathit{gen}\ (\theta_2\ \circ\ \theta_1\ \Gamma)\ \theta_2\ c_1])\ e_2 = (\theta_3, c)$, then $\mathit{assert}\ c\ e_2 = e_2$

Proof:

True by IH2, from which follows that if $\mathcal{CW}\ \Gamma\ (\textbf{let}\ x = e_1\ \textbf{in}\ e_2) = (\theta_3 \circ \theta_2 \circ \theta_1, c)$, then $\mathit{assert}\ c\ (\textbf{let}\ x = e_1\ \textbf{in}\ e_2) = (\textbf{let}\ x = e_1\ \textbf{in}\ e_2)$

Case $(x :: xs)$

To show: if $\mathcal{CW}\ \Gamma\ (x :: xs) = (\theta_2\ \circ\ \theta_1, \mathit{list}_i\ \texttt{<@>}\ c)$, then $\mathit{assert}\ (\mathit{list}_i\ \texttt{<@>}\ c)\ (x :: xs) = (x :: xs)$

IH1 If $\mathcal{CW}\ \Gamma\ x = (\theta_1, c)$, then $\mathit{assert}\ c\ x = x$

IH2 If $\mathcal{CW}\ \Gamma\ xs = (\theta_2, \mathit{list}_i\ \texttt{<@>}\ c)$, then $\mathit{assert}\ (\mathit{list}_i\ \texttt{<@>}\ c)\ xs = xs$

Proof:

  $\mathit{assert}\ (\mathit{list}\ \texttt{<@>}\ c)\ (x :: xs) = (x :: xs)$
$\equiv$ { Definition of <@> }
  $\mathit{assert}\ (\mathit{list}\ \&\ \mathit{Functor}\ c)\ (x :: xs) = (x :: xs)$
$\equiv$ { Definition of & }
  $(\mathit{assert}\ (\mathit{Functor}\ c) \circ \mathit{assert}\ \mathit{list})\ (x :: xs)$
$\equiv$ { Definition of *assert* and *list* }
  $\mathit{assert}\ (\mathit{Functor}\ c)\ (x :: xs)$
$\equiv$ { Definition of *assert* }

$$
\begin{aligned}
& \mathit{fmap}\ (\mathit{assert}\ c)\ (x :: \mathit{xs}) \\
\equiv\ & \{\text{ Definition of } \mathit{fmap} \ \} \\
& (\mathit{assert}\ c\ x :: \mathit{fmap}\ (\mathit{assert}\ c)\ \mathit{xs}) \\
\equiv\ & \{\text{ Apply IH1 } \} \\
& (x :: \mathit{fmap}\ (\mathit{assert}\ c)\ \mathit{xs}) \\
\equiv\ & \{\text{ Definition of } \texttt{<@>}, \&, \mathit{assert} \text{ and } \mathit{list} \text{ in IH2, then apply IH2 } \} \\
& (x :: \mathit{xs})
\end{aligned}
$$

from which follows that if $\mathcal{CW}\ \Gamma\ (x :: \mathit{xs}) = (\theta_2 \circ \theta_1, \mathit{list}_i\ \texttt{<@>}\ c)$, then $\mathit{assert}\ (\mathit{list}_i\ \texttt{<@>}\ c)\ (x :: \mathit{xs}) = (x :: \mathit{xs})$

Case $\mathit{Just}\ x$

To show: if $\mathcal{CW}\ \Gamma\ (\mathit{Just}\ x) = (\theta, \mathit{maybe}\ \texttt{<@>}\ c)$, then $\mathit{assert}\ (\mathit{maybe}\ \texttt{<@>}\ c)\ (\mathit{Just}\ x) = (\mathit{Just}\ x)$.

IH If $\mathcal{CW}\ \Gamma\ x = (\theta, c)$, then $\mathit{assert}\ c\ x = x$

Proof:

$$
\begin{aligned}
& \mathit{assert}\ (\mathit{maybe}\ \texttt{<@>}\ c)\ (\mathit{Just}\ x) \\
\equiv\ & \{\text{ Definition of } \texttt{<@>} \ \} \\
& \mathit{assert}\ (\mathit{maybe}\ \&\ \mathit{Functor}\ c)\ (\mathit{Just}\ x) \\
\equiv\ & \{\text{ Definition of } \& \ \} \\
& (\mathit{assert}\ (\mathit{Functor}\ c) \circ \mathit{assert}\ \mathit{maybe})\ (\mathit{Just}\ x) \\
\equiv\ & \{\text{ Definition of } \mathit{assert} \text{ and } \mathit{maybe} \ \} \\
& \mathit{assert}\ (\mathit{Functor}\ c)\ (\mathit{Just}\ x) \\
\equiv\ & \{\text{ Definition of } \mathit{assert} \ \} \\
& \mathit{fmap}\ (\mathit{assert}\ c)\ (\mathit{Just}\ x) \\
\equiv\ & \{\text{ Definition of } \mathit{fmap} \ \} \\
& \mathit{Just}\ (\mathit{assert}\ c\ x) \\
\equiv\ & \{\text{ Apply IH } \} \\
& \mathit{Just}\ x
\end{aligned}
$$

From which follows that if $\mathcal{CW}\ \Gamma\ (\mathit{Just}\ x) = (\theta, \mathit{maybe}\ \texttt{<@>}\ c)$, then $\mathit{assert}\ (\mathit{maybe}\ \texttt{<@>}\ c)\ (\mathit{Just}\ x) = (\mathit{Just}\ x)$.

Case (x, y)

To show: if $\mathcal{CW}\ \Gamma\ (x, y) = (\theta_2 \circ \theta_1, \mathit{pair}\ \texttt{<@@>}\ (c_1, c_2))$, then $\mathit{assert}\ (\mathit{pair}\ \texttt{<@@>}\ (c_1, c_2))\ (x, y) = (x, y)$.

IH1 If $\mathcal{CW}\ \Gamma\ x = (\theta_1, c_1)$, then $\mathit{assert}\ c_1\ x = x$

IH2 If $\mathcal{CW}\ \Gamma\ y = (\theta_2, c_2)$, then $\mathit{assert}\ c_2\ y = y$

Proof:

$$\begin{array}{ll}
 & \mathit{assert}\ (\mathit{pair} \mathbin{<@@>} (c_1, c_2))\ (x, y) \\
\equiv & \{\ \text{Definition of } \mathbin{<@@>}\ \} \\
 & \mathit{assert}\ (\mathit{pair}\ \&\ \mathit{Bifunctor}\ c_1\ c_2)\ (x, y) \\
\equiv & \{\ \text{Definition of } \&\ \} \\
 & (\mathit{assert}\ (\mathit{Bifunctor}\ c_1\ c_2) \circ \mathit{assert\ pair})\ (x, y) \\
\equiv & \{\ \text{Definition of } \mathit{assert} \text{ and } \mathit{pair}\ \} \\
 & \mathit{assert}\ (\mathit{Bifunctor}\ c_1\ c_2)\ (x, y) \\
\equiv & \{\ \text{Definition of } \mathit{assert}\ \} \\
 & \mathit{bimap}\ (\mathit{assert}\ c_1)\ (\mathit{assert}\ c_2)\ (x, y) \\
\equiv & \{\ \text{Definition of } \mathit{bimap}\ \} \\
 & (\mathit{assert}\ c_1\ x, \mathit{assert}\ c_2\ y) \\
\equiv & \{\ \text{Apply IH1, IH2}\ \} \\
 & (x, y)
\end{array}$$

from which follows that if $\mathcal{CW}\ \Gamma\ (x, y) = (\theta_2 \circ \theta_1, \mathit{pair} \mathbin{<@@>} (c_1, c_2))$, then $\mathit{assert}\ (\mathit{pair} \mathbin{<@@>} (c_1, c_2))\ (x, y) = (x, y)$.

Case *Left x*

To show: if $\mathcal{CW}\ \Gamma\ (\mathit{Left}\ x) = (\theta, \mathit{either} \mathbin{<@@>} (c, \mathit{true}))$, then $\mathit{assert}\ (\mathit{either} \mathbin{<@@>} (c, \mathit{true}))\ (\mathit{Left}\ x) = (\mathit{Left}\ x)$.

IH If $\mathcal{CW}\ \Gamma\ x = (\theta, c)$, then $\mathit{assert}\ c\ x = x$

Proof:

$$\begin{array}{ll}
 & \mathit{assert}\ (\mathit{either} \mathbin{<@@>} (c, \mathit{true}))\ (\mathit{Left}\ x) \\
\equiv & \{\ \text{Definition of } \mathbin{<@@>}\ \} \\
 & \mathit{assert}\ (\mathit{either}\ \&\ \mathit{Bifunctor}\ c\ \mathit{true})\ (\mathit{Left}\ x) \\
\equiv & \{\ \text{Definition of } \&\ \} \\
 & (\mathit{assert}\ (\mathit{Bifunctor}\ c\ \mathit{true}) \circ \mathit{assert\ either})\ (\mathit{Left}\ x) \\
\equiv & \{\ \text{Definition of } \mathit{assert} \text{ and } \mathit{either}\ \} \\
 & \mathit{assert}\ (\mathit{Bifunctor}\ c\ \mathit{true})\ (\mathit{Left}\ x) \\
\equiv & \{\ \text{Definition of } \mathit{assert}\ \} \\
 & \mathit{bimap}\ (\mathit{assert}\ c)\ (\mathit{assert\ true})\ (\mathit{Left}\ x) \\
\equiv & \{\ \text{Definition of } \mathit{bimap}\ \} \\
 & \mathit{Left}\ (\mathit{assert}\ c\ x) \\
\equiv & \{\ \text{Apply IH}\ \} \\
 & \mathit{Left}\ x
\end{array}$$

from which follow that if $\mathcal{CW}\ \Gamma\ (\mathit{Left}\ x) = (\theta, \mathit{either} \mathbin{<@@>} (c, \mathit{true}))$, then $\mathit{assert}\ (\mathit{either} \mathbin{<@@>} (c, \mathit{true}))\ (\mathit{Left}\ x) = (\mathit{Left}\ x)$.

Bibliography

[1] Design by Contract. In D Mandrioli and B Meyer, editors, *Advances in Object-Oriented Software Engineering*, pages 1–50. Prentice Hall, 1991.

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

[3] 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.

[4] Patrick Cousot and Radhia Cousot. Automatic Synthesis of Optimal Invariant Assertions: Mathematical Foundations. In *Symposium on Artificial Intelligence and Programming Languages*, 1977.

[5] Patrick Cousot, Radhia Cousot, Francesco Logozzo, and Michael Barnett. An Abstract Interpretation Framework for Refactoring with Application to Extract Methods with Contracts. In *OOPSLA'12*, October 2012.

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

[7] Michael D Ernst. *Dynamically Discovering Likely Program Invariants.* PhD thesis, University of Washington, 2000.

[8] Robert Bruce Findler and Matthias Felleisen. Contracts for Higher-Order Functions. In *ICFP'02*, October 2002.

[9] Alex Gerdes, Bastiaan Heeren, and Johan Jeuring. Properties of Exercise Strategies. In *IWS 2010*, pages 21–34, 2010.

[10] Alex Gerdes, Bastiaan Heeren, and Johan Jeuring. Teachers and students in charge - Using Annotated Model Solutions in a Functional Programming Tutor. In *EC-TEL '12*, pages 383–388, 2012.

[11] Alex Gerdes, Johan Jeuring, and Bastiaan Heeren. An Interactive Functional Programming Tutor. In *ITiCSE '12*, pages 250–255, 2012.

[12] Bastiaan Heeren, Johan Jeuring, and Alex Gerdes. Specifying Rewrite Strategies for Interactive Exercises. *Mathematics in Computer Science*, 3:349–370, March 2010.

[13] Ralf Hinze, Johan Jeuring, and Andres Löh. Typed contracts for functional programming. In *FLOPS'06*, pages 208–225, 2006.

[14] C A R Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, October 1969.

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

[16] Bertrand Meyer. Applying 'Design by Contract'. *Companion of the 13th international conference*, 25:40–51, October 1992.

[17] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, December 1978.

[18] Simon Peyton Jones. Haskell 98 Language and Libraries. *Journal of Functional Programming*, 13(1), January 2003.

[19] J A Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 1965.

[20] Michael Sperber, R Kent Dybvig, Matthew Flatt, and Anton van Straaten. Revised6 Report on the Algorithmic Language Scheme. Technical report, September 2007.

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

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

[23] Dana N Xu, Simon Peyton Jones, and Koen Claessen. Static Contract Checking for Haskell. In *POPL'09*, pages 1–12, January 2009.