Department of Computing Science

MSc Thesis, ICA-3682536

Better Feedback for Incorrect Programs by Using Contracts

Author:
Jacco Krijnen

Supervisor:
Prof. dr. J.T. Jeuring

August 21th 2016

Abstract

Ask-Elle is a programming tutor that aims to help students learn the Haskell programming language. It does this by providing programming exercises and assisting the student during the process. The tutor can automatically provide hints to a solution when the student gets stuck or give counterexamples for incorrect submissions. It is these incorrect programs that we are interested in. The goal of this thesis is to improve the detail of feedback by automatically monitoring contracts on parts of the program. When running it with the counter example, the tutor should be able to explain more clearly what the mistake might be. We analyze a previous approach that mixes type inference with contracts and find that this is not always correct. We then formalize the problem statement and define a criterion for sound ways of monitoring an incorrect program. Finally, we have implemented the system in Haskell.

Preface

My passion for functional programming originated in 2012, several months after finishing the introductory course *functioneel programmeren*. This first encounter with Haskell was no "love at first sight": I had just finished my first year with good results and had learned the skill of programming. The year had revolved around one imperative programming language which shaped my way of thinking. I remember the joyous feeling of having no boundaries at all as for what I could create.

Along came the second year and Haskell hit me in the face. This was a fundamentally different way of expressing your thoughts to the computer and felt all but intuitive. The claims about conciseness, types and purity made no sense to me, I already knew how to program. And why did we have to work with a plain text editor after having spent months in a highly sophisticated IDE? I think my affinity with logic and formalisms kept me somewhat interested.

After finishing the course I quickly went back to the imperative world and forgot about Haskell. As part of an extracurricular activity a few months later, I found myself working on an assignment about interleaving parsers with Doaitse Swierstra. It was then that I started to appreciate the expressivity of Haskell. I was invited to assist at the summerschool for functional programming and this is when my interest quickly started growing.

I noticed that there were many other students that had a similar first experience with functional programming and just gave up. Although the structure of the curriculum might have been partly to blame, there remains the fact that Haskell tooling is very minimalistic when compared to IDE's like *Visual Studio* for C# or *IntelliJ* for Java. The simple explanation is that these languages are maintained by multibillion-dollar companies backing its ecosystem and tooling. Haskell on the other hand has its roots in the much smaller academic world of programming language design. In recent years the language is also extending towards the world of software industry. Subsequently several initiatives [1] have as their goal to improve the tooling and ecosystem as well as offering consultancy for companies using or wanting to use Haskell.

This thesis is an attempt to contribute to a better learning experience for beginner Haskell programmers by using automated tooling.

I would like to thank my supervisor Johan Jeuring for the time he took in helping me realise this thesis. Without the countless meetings, discussions at the whiteboard, feedback and pointers to literature I would have never managed to come to this result. Furthermore I want to thank Wouter Swierstra, Jurriaan Hage and Ruud Koot for some insightful conversations. Finally, I'd like to thank my parents who were always willing to listen to me when I was stuck, even though most problems were of a technical nature that went far over their heads.

[1] FP Complete, Industrial Haskell Group, Well-Typed among others

Contents

1 Introduction **4**
- 1.1 Ask-Elle . . . 4
- 1.2 Contracts . . . 5

2 Syntax and semantics **7**
- 2.1 The language λ_+ . . . 7
- 2.2 The language of contracts . . . 7
- 2.3 Operational semantics of λ_+ . . . 9
- 2.4 Reasoning about monitoring . . . 12

3 Contract Inference **15**
- 3.1 The Hindley-Milner type system . . . 16
- 3.2 User defined contracts or refinements . . . 16
- 3.3 Correctness properties for an inference algorithm . . . 20
- 3.4 Refinement contracts and strengthening . . . 21
- 3.5 Type variables and contracts . . . 21
- 3.6 Summary . . . 23

4 Contract Propagation and Folds **24**
- 4.1 Folds . . . 24
- 4.2 Preserving properties . . . 24
  - 4.2.1 An annotator for `foldr` . . . 26
- 4.3 Dependent contracts . . . 28
- 4.4 On the definition of a consistent annotator . . . 28
- 4.5 Propagation in other calls to foldr . . . 29
- 4.6 Generalizing the annotator . . . 29

5 Related work and future research **31**
- 5.1 Liquid types . . . 31
- 5.2 Contracts and laziness . . . 31
- 5.3 Mechanizing the formalization . . . 32
- 5.4 Improving location information . . . 32
- 5.5 Taking contract propagating further . . . 33
- 5.6 Debuggers . . . 33

6 Conclusion **34**

A Consistency of dependent contract annotator **35**

B Implementation details **37**

Chapter 1

Introduction

Suppose you were asked to write a function that sorts a list of integers. How many correct solutions could you come up with? One could first choose between different sorting strategies, for example mergesort or quicksort. But when writing actual code, it is possible to come up with infinitely many programs that differ syntactically but are all behaviorally equivalent and all sort a list of integers.

Equally, there are infinitely many *incorrect* solutions to the sorting problem. It is those programs which do *not* implement their specification that we are interested in. We want to explore what techniques can be used to automatically give feedback on mistakes in such code.

In this thesis, we describe an approach for generating better feedback by using contracts. To understand the title, we describe the two main ingredients: Ask-Elle and contracts.

1.1 Ask-Elle

Ask-Elle [15] is a programming tutor targeting bachelor students starting to learn the Haskell programming language. It is a web-based application developed by Johan Jeuring, Alex Gerdes and Bastiaan Heeren. The term "tutor" comes from several features it has:

Incremental programming The student can submit programs that are not finished by writing the `?` symbol for subexpressions or patterns that are yet to be implemented.

Feedback on whether the student is on the right track The tutor tracks whether the student is on the right path by comparing the current solution with several model solutions.

Hints on how to continue when the student is stuck By having these model solutions, the tutor can explain what steps to take to go in the right direction from a partial program.

Show counterexamples When the tutor cannot match the submitted program, it tests the specification by using quickcheck. When a counter example is found it is presented to the student

Currently, the feedback for incorrect programs is very minimal and the student is left on his own to decide what the actual mistake is. When faced with a program that does not meet its specification, we can try to generate answers to different questions. For example

- In **what cases** does the program behave incorrectly?
- **Why**, in terms of the specification, is the program incorrect?
- **Where** in the source code is a mistake?

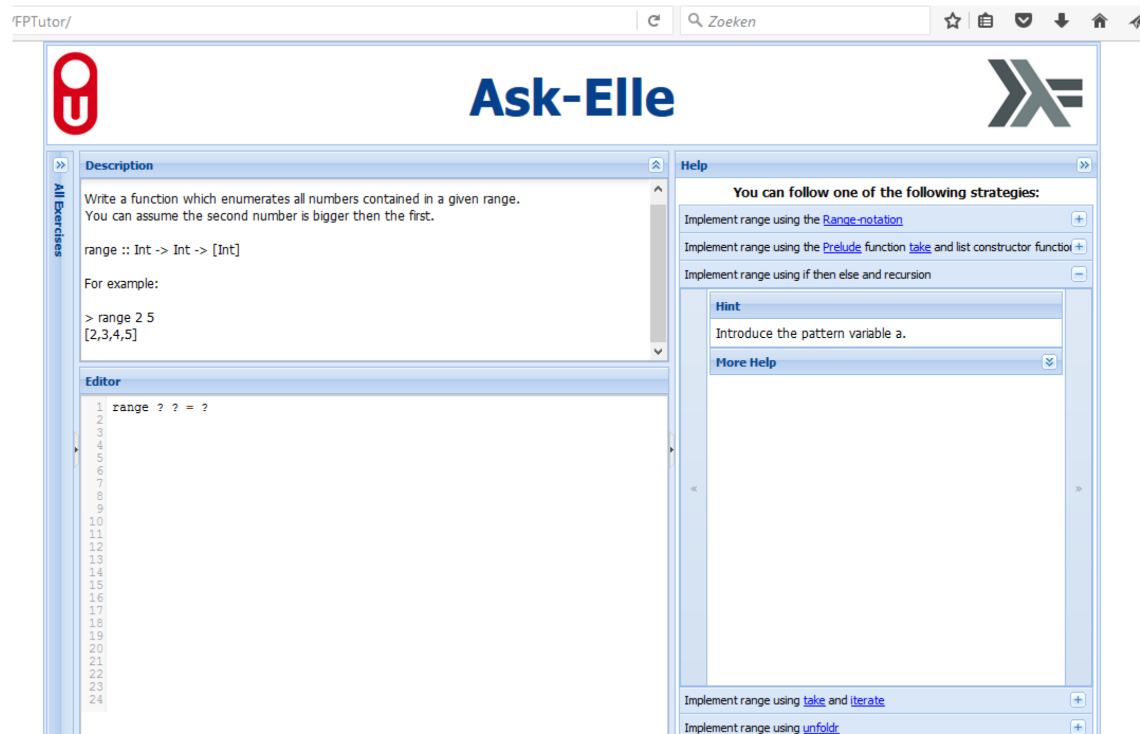

Figure 1.1: The Ask-Elle tutor

Naturally, a static type system detects many incorrect programs by checking the types of a program. For example, a program that squares its integer input is of types $Int \rightarrow Int$, whereas any sorting function should have the type $[Int] \rightarrow [Int]$. Such type errors are caught at compile-time and are of course utilized by Ask-Elle. It answers the first and third question for some cases. This technique cannot distinguish between programs with the same type, however. Quickcheck can provide us with counterexamples for an invalid solution, answering "what cases" in a more detailed way than a type system. We believe that by expressing the exercise specification as a *contract* we can also show more precisely *where* and more importantly *why* a program is incorrect.

1.2 Contracts

A *Contract* specifies conditions and obligations for a function, similar to a contract in the real world. For example, a contract might specify that a function requires an even integer and will return a prime number greater than the provided integer. Contracts can be *violated* when a function receives an incorrect value or produces an incorrect value. A violation will usually raise an exception, giving an indication which function is to *blame*.

This mechanism is the basis for the design-by-contract [20], which was made popular by the Eiffel language in the 1980's [19].

Haskell also has contracts, for example the `typed-contracts` aproach [13] defines them as follows:

```
data Contract a where
    Prop      :: (a -> Bool) -> Contract a
    Function  :: Contract a  -> (a -> Contract b) -> Contract (a -> b)
    List      :: Contract a  -> (Contract [a])
    ...
```

This defines several contructors. The first, `Prop`, takes a predicate to denote a simple property of a value. The `Function` constructor takes a contract for the argument value and a contract for the result value (which may depend on the argument) and produces a contract for function types. Furthermore, the library contains useful constructs for other datatypes and typeclasses such as lists, pairs and functors.

One can now write contracts such as

```
evenInPrimeOut :: Contract (Int -> Int)
evenInPrimeOut =
    Function (Prop (\x -> even x))
             (\x -> (Prop (\y -> isPrime y && y > x)))
```

Which describes a function that takes an even integer and returns a prime number that is greater than the provided even integer. The contract can be monitored[1] on any function of the type.

```
f :: Int -> Int
f = monitor evenInPrimeOut $ \x -> x*x + 1
```

That is, `monitor` has type $Contract\ a \to a \to a$ and crashes with an exception whenever the contract is violated.

[1] we write `monitor` to be consistent with the rest of this thesis, the actual library function is called `assert`

Chapter 2

Syntax and semantics

In this chapter we will introduce λ_+, the programming language that will be used for the rest of the thesis. It is based on the lambda calculus, extended with some useful constructs such basic datatypes, pattern matching and contract monitoring. We give a big-step call-by-value semantics in the style of [22] that we use to define semantic equality of terms and prove a useful lemma for Chapter 4. Furthermore, we describe the language of contracts and discuss the semantics of contract monitoring.

2.1 The language λ_+

Figure 2.1 shows the syntax of λ_+. It includes the usual constructs of variables, function application and lambda abstraction and a let-binding. Furthermore, we have boolean and numeric literals, list constructors, common operators and pattern matching. We make a distinction between terms and annotated terms. The reason for this is that the **monitor** construct is not meant to be available to the programmer, but is only inserted for contract checking purposes. We use $\uplus$ to indicate an extension to an existing grammar. We will discuss the language of contracts τ in the next section. For convenience we might write t to range over annotated terms as well when the context is clear.

We will use the syntactical operation **forget** : $t^+ \to t$ which recursively removes all **monitor** annotations from an annotated term, i.e. **forget**(**monitor** $c\ t$) = **forget**(t), and for all other terms recursively removes annotations in subterms. Furthermore, we write substitution as $t[t'/x]$ which recursively replaces all free occurrences of x by t' (it also replaces variables in terms that occur as part of a contract). We consider a top-level definition like $f = t$ to be syntactic sugar for **let** $f = t$ **in** f.

We think that this language is a suitable mid-point between the pure lambda calculus and full Haskell: the extra constructs allow us to write some realistic programs that can occur in Ask-Elle, while keeping the language small enough to make the reasoning not too cumbersome.

2.2 The language of contracts

We present a language of contracts that is very similar to the *types* in a Hindlley-Milner setting, as opposed to the usual EDSL notation in Haskell. The reason to do so is to have a convenient notation when we will discuss type inference algorithms in Chapter 3. Although it is common practice in Haskell to define contracts as an EDSL [13], we can see that any such contract can also be described using this language, as predicates over basetypes appear as refinements and all the types of the typed contracts become explicit as τ_{base} in the refined basetype.

Figure 2.2 shows the syntax of contracts. The third production of τ shows that any basetype is decorated with a refinement (a predicate) that limits the set of values indicated by that basetype. For example $\{\texttt{x} : Int \mid \texttt{x>=0}\}$ indicates the type of natural numbers. These kind of types are often

Basic Term

$t ::= \texttt{x}$	variable
$\mid \lambda\texttt{x}.t$	lambda abstraction
$\mid t\ t$	application
$\mid \textbf{let}\ x = t\ \textbf{in}\ t$	let binding
$\mid t \oplus t$	$\oplus \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{\&\&}, \texttt{\|\|}, \texttt{==}, \texttt{<}, \texttt{>}\}$, binary operator
$\mid \textbf{case}\ t\ \textbf{of}$	
$p_1 \rightarrow t_1$	
$\ldots$	
$p_n \rightarrow t_n$	case distinction
$\mid n$	$n \in \mathbb{Z}$, numeric literal
$\mid \texttt{True} \mid \texttt{False}$	Boolean literal
$\mid t{:}t \mid \texttt{[]}$	list constructors

Annotated Term

$t^+ = t\ \uplus$	
$\mid \textbf{monitor}\ \tau\ t$	contract monitoring

Pattern

$p ::= \texttt{x}$	variable
$\mid \texttt{[]}$	empty list
$\mid p : p$	non-empty list

Figure 2.1: The syntax of λ_+

referred to as *refinement types*. Instead of $\{\texttt{x} : \tau \mid \texttt{True}\}$, we might write $\texttt{x} : \tau$, or just τ, depending on whether we want to bind x or not. Similarly, we might write $\{\tau \mid \rho\}$ if we do not want to name the value. Finally we write $\top$ for $\{\texttt{x} : \tau \mid \texttt{True}\}$ when the type τ is clear from the context.

Similarly to [24] we use the notation $\Gamma \vdash \sigma$ in Figure 2.3 to indicate that contract schema σ is well-formed in an enviroment Γ that binds variables to types. A well-formed contract should only have refinements that are correctly typed ([Refined-basetype]). Note that this premise is an actual type judgement from our implicit type system for $\lambda+$. Furthermore variables bound in refined base-types should be correctly scoped for dependent contracts ([Refined-basetype], [Function-Base]). In the case of a higher order function ([Function-HO]), the domain is not named and therefore the environment Γ is not extended.

Well-formed contract schemes allow for so-called dependent contracts that can state relations between input and output. For example, $\{\texttt{x} : \mathit{Int} \mid \texttt{x >= 0}\} \rightarrow \{\texttt{y} : \mathit{Int} \mid \texttt{y > x}\}$ describes a function that expects a natural number and returns an integer that is larger than the provided integer.

Contract

	$\tau ::= a, b, c$	type variable
	$\mid \tau \rightarrow \tau$	function
	$\mid \{\mathtt{x} : \tau_{base} \mid \rho\}$	refined basetype

Base type

$$\tau_{base} ::= \text{Bool} \mid \text{Int} \mid \ \ldots$$

Refinement

	$\rho ::= \alpha, \beta, \gamma$	Refinement variable
	$\mid t$	term

Contract scheme

	$\sigma ::= \forall a.\sigma$	Polymorphic binding
	$\mid \tau$	monomorphic type

Figure 2.2: The syntax of contracts

2.3 Operational semantics of λ_+

In Figure 2.4 we define a big step call-by-value operational semantics for closed annotated terms.

The big-step semantics are based on those of a call-by value lambda calculus. The judgements come in the form $t \Downarrow_m v$, which we read this as "Term t evaluates to value v with monitoring result m". First we define the subset of terms that constitute values. A monitoring result is either no violation ✓ or a violation with a specific message $\lightning s$. The operator $\triangleleft$ is a left biased choice, used to collect violation results. Note that we only define evaluation for closed terms by means of substitution, thus not requiring a rule for variables. Furthermore, we omit the semantics of the binary operators, as these follow their usual primitive implementation.

The [Value] rule states that any term t that is a value evaluates to itself with no violations. The [App] rule has the usual evaluation semantics with β -substitution with violation result $m_1 \triangleleft m_2 \triangleleft m_3$ (we will elaborate on this violation part of the semantics in the next subsection). Note that subtitution is also performed in the refinement terms of contracts, for example in rule [Mon-2] and [Mon-3]. This ensures that refinements of dependent contracts can use the actual arguments for their free variables. The [Letrec] rule allows recursive function bindings (other types of values will diverge). The [Case-*] rules describe pattern matching that, similar to [App], "binds" the variables by substitution. Finally, the [Mon-*] rules describe the process of contract monitoring. [Mon-1] and [Mon-2] evaluate the refinement (which on itself cannot cause violation, hence the ✓). The [Mon-3] rule describes the case of a refinement variable, in which case no checks are performed as such contracts indicate unconstrained types. The most interesting rule [Mon-4] gives the semantics for monitoring a dependent function contract. Such an expression evaluates to another function, which will monitor its domain with contract τ_1 in the **case** and monitor the result t' with contract τ_2. Note that a refinement in τ_2 may mention the variable x, which is therefore bound on the term level as x as well.

Note that the $\Downarrow$ relation on $t^+ \times (m \times v)$ is a partial function (modulo alpha conversion). This follows from the fact that the rules are syntax-directed: for every term at most one rule applies.

Well-formed contract scheme $\Gamma \vdash \sigma$

$$\frac{\Gamma \vdash \mathtt{t} : Bool \qquad \mathbf{fv}(t) \subseteq \mathbf{vars}(\Gamma)}{\Gamma \vdash \{\mathtt{x} : \tau_{base} \mid \mathtt{t}\}} \text{[Refined-basetype]}$$

$$\frac{}{\Gamma \vdash \{\mathtt{x} : \tau_{base} \mid \alpha\}} \text{[Refinement-variable]}$$

$$\frac{\Gamma \vdash \tau \qquad \Gamma, \mathtt{x} : \tau_{base} \vdash \tau' \qquad \tau = \{\mathtt{x} : \tau_{base} \mid \rho\}}{\Gamma \vdash \tau \to \tau'} \text{[Function-Base]}$$

$$\frac{\Gamma \vdash \tau \qquad \Gamma \vdash \tau' \qquad \tau \neq \{\mathtt{x} : \tau_{base} \mid \rho\}}{\Gamma \vdash \tau \to \tau'} \text{[Function-HO]}$$

$$\frac{\Gamma \vdash \sigma}{\Gamma \vdash \forall a.\sigma} \text{[Contract-schema]}$$

Figure 2.3: Well-formed contract schemes

Monitoring results and exception semantics

In practice, most contract library implementations [6,13] implement contract violations in the form of *exceptions*. That is, the program halts as soon as the contract is found out to be violated. Since usual exception semantics are quite cumbersome to formalize with big-step semantics (resulting in a "duplication problem" [3]) we choose a different approach to keep the semantics as simple as possible: a contract violation will not halt execution. Rather, the program silently collects violations and combines them using the $\triangleleft$ operator. This operator is left-biased, so the argument order matters. In general, the order of arguments for $\triangleleft$ in the rules of figure 2.4 follows the order of evaluation, effectively collecting the first violation encountered by evaluation. For example, in [Case-1] the message resulting from t has priority over the message from t_1. By choosing violation messages in this way, we obtain the same message that an exception mechanism would give us. The [App] and [Cons] rules are the only rules that do not specify the order of evaluation of their subterms, as is common for big-step semantics. We therefore arbitrarily choose the first term to have precedence.

Incorporating a complete exception semantics would also be unnecessary as we do not need a catching mechanism (contract violations are not observable for the programmer). Another difference with this approach is that for diverging programs, we cannot obtain a contract violation, whereas exception semantics might have stopped the program execution when the violation was raised. We argue that for our purposes this is no problem: Ask-Elle only specifies terminating functions. In case a student accidentally produces a non-terminating program, Ask-Elle will give a time-out message.

An advantage of this approach is that by introducing a different collecting semantics we could technically collect all contract violations, just like a compiler collects multiple type-errors. However, for the scope of the Ask-Elle tutor, it is not necessary to present the student with multiple messages. Furthermore, some messages might not make sense since they are consequences of other violations, for example a postcondition that relies on a precondition. In compiler error messages there is not always a clear error that comes first and presenting the errors in the right order is non-trivial [11].

Value v

$$\frac{n \in \mathbb{Z}, m \in \textbf{String}}{\lambda\texttt{x . } t, n, \texttt{True}, \texttt{False}, \texttt{[]} \text{ are values}}$$

$$\frac{t_1 \text{ and } t_2 \text{ are values}}{t_1 : t_2 \text{ is a value}}$$

Monitoring result m

$$\begin{aligned} m &::= \checkmark && \text{no violation} \\ &\mid \lightning\ s && s \in \textbf{String}\text{, contract violation} \end{aligned}$$

$$m_1 \lhd m_2 = \begin{cases} m_1 & \text{if } m_1 = \lightning\, s \\ m_2 & \text{otherwise} \end{cases}$$

Strict evaluation $t \Downarrow_m v$

$$\frac{t \text{ is a value}}{t \Downarrow_{\checkmark} t} \text{[Value]}$$

$$\frac{t_1 \Downarrow_{m_1} v_1 \qquad t_2 \Downarrow_{m_2} v_2}{t_1 : t_2 \Downarrow_{m_1 \lhd m_2} v_1 : v_2} \text{[Cons]}$$

$$\frac{t_1 \Downarrow_{m_1} \lambda\texttt{x . } t \qquad t_2 \Downarrow_{m_2} v_1 \qquad t[v_1/x] \Downarrow_{m_3} v_2}{t_1 t_2 \Downarrow_{m_1 \lhd m_2 \lhd m_3} v_2} \text{[App]}$$

$$\frac{t_1[\textbf{let } x = t_1 \textbf{ in } x/x] \Downarrow_{m_1} v_1 \qquad t_2[v_1/x] \Downarrow_{m_2} v_2}{\textbf{let } x = t_1 \textbf{ in } t_2 \Downarrow_{m_1 \lhd m_2} v_2} \text{[Letrec]}$$

$$\frac{t \Downarrow_{m_1} v_1 \qquad p \text{ unifies with } v_1 \text{ by substitutions } \theta \qquad \theta t_1 \Downarrow_{m_2} v_2}{\textbf{case } t \textbf{ of } \{p \to t_1; ps\} \Downarrow_{m_1 \lhd m_2} v_2} \text{[Case-1]}$$

$$\frac{t \Downarrow_{m_1} v_1 \qquad p, v_1 \text{ do not unify} \qquad \textbf{case } v_1 \textbf{ of } ps \Downarrow_{m_2} v_2}{\textbf{case } t \textbf{ of } \{p \to t_1; ps\} \Downarrow_{m_1 \lhd m_2} v_2} \text{[Case-2]}$$

$$\frac{t_2 \Downarrow_m v \qquad t_1[v/x] \Downarrow_{\checkmark} \texttt{True}}{\textbf{monitor } \{x : _ \mid t_1\}\ t_2 \Downarrow_m v} \text{[Mon-1]}$$

$$\frac{t_2 \Downarrow_{m_2} v \qquad t_1[v/x] \Downarrow_{\checkmark} \texttt{False} \qquad s \text{ is the violation message}}{\textbf{monitor } \{x : _ \mid t_1\}\ t_2 \Downarrow_{\lightning s} v} \text{[Mon-2]}$$

$$\frac{t \Downarrow_m v}{\textbf{monitor } \{x : _ \mid \alpha\}\ t \Downarrow_m v} \text{[Mon-3]}$$

$$\frac{t \Downarrow_m \lambda x.t'}{\textbf{monitor } ((x : \tau_1) \to \tau_2)\ t \Downarrow_m \lambda x.\textbf{case } (\textbf{monitor } \tau_1\ x) \textbf{ of } x \to \textbf{monitor } \tau_2\ t')} \text{[Mon-4]}$$

Figure 2.4: Big step call-by-value evaluation of λ_+

The violation message

For violations to have some actual use to the programmer, we need an accompanying message. The [Mon-2] rule has the rather abstract premise: "s is the violation message". This is on purpose to keep the semantics as simple as possible. Furthermore, there are different ways to produce a useful message. In any case, the message should depend on the contract being monitored. For example by giving well-formed contracts a bit more structure: $\Gamma \vdash \{x : \tau_{base} \mid t \mid s\}$ where s is a **String**-like expression that can contain free variables such that $\mathbf{vf}(s) \subseteq \mathbf{vars}(\Gamma)$. Lauwers [16] goes even further by taking the syntax tree and line/columnn information into account when generating these messages. Yet another way is by relying on the operational semantics to provide useful location information such as a stack trace [1].

Call-by-value

We chose a strict semantics for our language to simplify the operational semantics and the reasoning involved in the next chapters. Although Ask-Elle obviously runs Haskell, we justify this simplification for now by the fact that the nature of the exercises is to introduce functional programming, and not so much lazy semantics. We will go into more depth on the required changes for lazy semantics in chapter 5 on future research.

2.4 Reasoning about monitoring

For the rest of this thesis, we will use the following notation:

1. $t\lightning$ means that $t \Downarrow_{\lightning s} v$ for some violation message s and value v. We say that t *raises a violation*
2. $t\checkmark$ means $t \Downarrow_{\checkmark} v$ for some value v. We say that t is *violation free*

In chapter 4, we will prove some properties of certain program transformations. In the rest of this section we define semantic equality in terms of the operational semantics and prove a simple Lemma that we will use later on.

Definition 2.4.1. Values v and v' are semantically equal, written $v \equiv v'$, when either

- $v = v'$ (syntactic equality modulo alpha conversion)
- or $v = v_1 : v_2$ and $v' = v_3 : v_4$ and $v_1 \equiv v_3$ and $v_2 \equiv v_4$
- or $v = \lambda x.t_1$ and $v' = \lambda y.t_2$ and $\forall t.\ t_1[t/x] \equiv t_2[t/y]$ (extensional equality)

Definition 2.4.2. Terms t and t' are semantically equal, written $t \equiv t'$ if

$$t \Downarrow_m v \wedge t' \Downarrow_m v' \Rightarrow v \equiv v'$$

Note that the above definitions are mutually recursive: in the case that we consider value equality on lambda's we need term equality and for terms we evaluate and require again value equality. The definitions are not cyclic however, one can see this by considering the type of a lambda value: $\tau_1 \to \tau_2$, of which τ_2 indicates the type of the body. Although τ_2 can be a term that evaluates to another lamda, types are *finite* meaning that eventually the body of a lambda will evaluate to a value that is not a lambda. So in all cases we have a "finite" amount of mutual recursion between the two definitions.

As an example of semantic equivalence, consider $2 + 2 \Downarrow_{\checkmark} 4$. By the [Value] rule, we have $4 \Downarrow_{\checkmark} 4$ and thus $2 + 2 \equiv 4$. This principle can be generalized to all judgements: $t \Downarrow_{\checkmark} v$ implies $t \equiv v$. In proofs we will refer to these kind of equivalences as Eval-[Rulename].

From many of the rules we can also obtain some other useful equivalences. For example, if we have $t_1\ t_2 \Downarrow_{\checkmark} v$ then we know, because the rules are syntax-directed, that there exists a t such

that $t_1 \equiv \lambda x.\ t$ and v_1 such that $t_2 \equiv v_1$ and $t_1\ t_2 \equiv t[v_1/x]$. We will refer to such equivalences as [Rulename].

We now lay out some groundwork for proving in equational reasoning style. First, we observe that $\equiv$ forms an equivalence relation on values and on terms, so we can write proofs in the form $t_1 \equiv \cdots \equiv t_n$. Furthermore note that equality on the syntactic level ($x = y$) implies semantic equality ($x \equiv y$), we will explicitly write $=$ for reasoning at the meta-level during equational reasoning proofs.

Lemma 2.4.1. (Substitution lemma) $\forall t, t', v, x.\ t \Downarrow_m v \Rightarrow t'[t/x] \equiv t'[v/x]$

Proof By induction over t'. In the case that $t' = x$, we directly see that $x[t/x] \equiv t \equiv v \equiv x[v/x]$. In case $t' = y$ (y being a variable) and $x \neq y$ it is equally simple to see. In all other language constructs we can use the induction principle to assume equivalent subterms that therefore evaluate to equivalent values. □

Lemma 2.4.2. (General substitution lemma) $\forall t_1, t_2, t_3, x.\ t_1 \equiv t_2 \Rightarrow t_3[t_1/x] \equiv t_3[t_2/x]$

Proof We first prove for the case that t_1 and t_2 are values, i.e. $v_1 = t_1, v_2 = t_2$

- If $v_1 = v_2$, then $t_3[v_1/x] = t_3[v_2/x]$ and semantic equivalence follows directly.
- If $v_1 = v_{11} : v_{12}$ and $v_2 = v_{21} : v_{22}$ then by definition we have $v_{11} \equiv v_{21} \wedge v_{12} \equiv v_{22}$, and inductively $t_1[y : z/x][v_{11}/y][v_{12}/z] \equiv t_1[y : z/x][v_{21}/y][v_{22}/z]$
- If $v_1 = \lambda x.t$ and $v_2 = \lambda x.t'$, observe that in both derivation trees of $t_3[v_1/x]$ and $t_3[v_2/x]$ the only rule that will evaluate t and t' is [App]. The third premise of that rule asserts $t[v_1/x]$ and $t'[v_1/x]$ respectively. By the definition of semantic equivalence on functions, these are so too.

In the case that t_1 and t_2 are not values, note that there exist values $v_1 \equiv v_2$ such that $t_1 \Downarrow_m v_1 \wedge t_2 \Downarrow_m v_2$, by definition of equivalence on terms. We can then invoke the substitution lemma to prove:

$$\begin{aligned} & t_3[t_1/x] \\ \{\text{substitution lemma}\} \equiv\ & t_3[v_1/x] \\ \{\text{above reasoning}\} \equiv\ & t_3[v_2/x] \\ \{\text{substitution lemma}\} \equiv\ & t_3[t_2/x] \end{aligned}$$

□

The following lemma is very useful for equational reasoning, we will use it for terms with free variables, which are implicitly defined. For example, it is common to reason about a term t containing `foldr` as if it was **let** `foldr` $= \ldots$ **in** t

Lemma 2.4.3. (Replacing a variable by its definition) ($\textbf{let}\ x = t_1\ \textbf{in}\ t_2$) $\equiv$ ($\textbf{let}\ x = t_1\ \textbf{in}\ t_2[t_1/x]$)

Proof By the [Letrec] rule there exists a value v_1 such that $t_1[\textbf{let}\ x = t_1\ \textbf{in}\ x/x] \equiv v_1$ and $\textbf{let}\ x = t_1\ \textbf{in}\ t_2 \equiv t_2[v_1/x]$.

$$\begin{aligned} & \textbf{let}\ x = t_1\ \textbf{in}\ t_2 \\ \{[\text{Letrec}]\} \equiv\ & t_2[v_1/x] \\ \{[\text{Letrec}]\} \equiv\ & t_2[t_1[\textbf{let}\ x = t_1\ \textbf{in}\ x/x]/x] \\ \{a[b[c/x]/x] = a[b/x][c/x]\} =\ & t_2[t_1/x][\textbf{let}\ x = t_1\ \textbf{in}\ x/x] \\ \{[\text{Letrec}]\} \equiv\ & \textbf{let}\ x = t_1\ \textbf{in}\ t_2[t_1/x] \end{aligned}$$

□

We now prove two properties specific to contract monitoring.

Lemma 2.4.4. $(\textbf{monitor}\ (\top \to \varphi)\ \ t_1)\ t_2 \equiv \textbf{monitor}\ \varphi\ \ (t_1\ \ t_2)$

Proof By [Mon-4], there exist x, t' such that $t_1 \equiv \lambda x.\ t'$

$$\begin{aligned}
& (\textbf{monitor}\ (\top \to \varphi)\ \ t_1)\ t_2 \\
\{\text{gen. subst.}\} \equiv\ & (\textbf{monitor}\ (\top \to \varphi)\ \ (\lambda x.\ \ t'))\ t_2 \\
\{\text{Eval-[Mon-4]}\} \equiv\ & (\lambda x.\textbf{case}\ (\textbf{monitor}\ \top\ \ x)\ \textbf{of}\ x \to \textbf{monitor}\ \varphi\ t')\ t_2 \\
\{\text{Eval-[Mon-1] \& gen. subst}\} \equiv\ & (\lambda x.\textbf{case}\ x\ \textbf{of}\ x \to \textbf{monitor}\ \varphi\ t')\ t_2 \\
\{\text{[Case-1]}\} \equiv\ & (\lambda x.\textbf{monitor}\ \varphi\ t'[x/x])\ t_2 \\
=\ & (\lambda x.\textbf{monitor}\ \varphi\ t')\ t_2 \\
\{\text{[App]}\} \equiv\ & \textbf{monitor}\ \varphi\ t'[t_2/x] \\
\{\text{[App]}\} \equiv\ & \textbf{monitor}\ \varphi\ ((\lambda x.t')\ t_2) \\
\{\text{gen. subst}\} \equiv\ & \textbf{monitor}\ \varphi\ (t_1\ t_2)
\end{aligned}$$

Lemma 2.4.5. $\forall t.\ t\checkmark \Rightarrow t \equiv \textbf{forget}(t)$

Proof By induction on t. Assume $t \Downarrow_{\checkmark} v$ and $\textbf{forget}(t) \Downarrow_m v'$. We present only a few cases to show the essence of the proof:

- $t = \textbf{monitor}\ c\ \ t_1$

  By assumption we have a derivation for $\textbf{monitor}\ c\ t_1 \Downarrow_{\checkmark} v$ ending in one of the [Mon-*] rules which always includes a premise that evaluates t_1:

  $$\frac{\cdots \qquad t_1 \Downarrow_{\checkmark} v \qquad \cdots}{\textbf{monitor}\ c\ t_1 \Downarrow_{\checkmark} v}\ \text{[Mon-*]}$$

  By applying the induction hypothesis on t_1, we get $\textbf{forget}(t_1) \Downarrow_{\checkmark} v$. By the definition of **forget** we have that $\textbf{forget}(t_1) = \textbf{forget}(\textbf{monitor}\ \ c\ \ t_1)$ and thus $\textbf{forget}(\textbf{monitor}\ \ c\ \ t_1) \Downarrow_{\checkmark} v$. By assumption, we have that $\textbf{forget}(\textbf{monitor}\ \ c\ \ t_1) \Downarrow_m v'$ and since $\Downarrow$ is a partial function, we have $m = \checkmark$ and $v = v'$ (so $v \equiv v'$).

- $t = t_1\ t_2$

  Consider $\textbf{forget}(t) = \textbf{forget}(t_1\ \ t_2) = \textbf{forget}(t_1)\ \ \textbf{forget}(t_2)$. By assumption we have two derivations ending in the [App] rule:

  $$\frac{t_1 \Downarrow_{m_1} \lambda x.\ t_3 \qquad t_2 \Downarrow_{m_2} v_1 \qquad t_3[v_1/x] \Downarrow_{m_3} v_2}{t_1\ t_2 \Downarrow_{\checkmark} v_2}\ \text{[App]}$$

  $$\frac{\textbf{forget}(t_1) \Downarrow_{m_1'} \lambda x'.\ t_3' \qquad \textbf{forget}(t_2) \Downarrow_{m_2'} v_1' \qquad t_3'[v_1'/x] \Downarrow_{m_3'} v_2'}{\textbf{forget}(t_1)\ \textbf{forget}(t_2) \Downarrow_{m_1' \triangleleft m_2' \triangleleft m_3'} v_2'}\ \text{[App]}$$

  First note that any term that evaluates to a value and contains no **monitor** construct will always be violation free: with a simple inductive argument over the rules without [Mon-*] we see that $\lightning s$ is never introduced, only [Value] introduces a $\checkmark$, and any of the other rules combine using $\checkmark \triangleleft \checkmark = \checkmark$. So we have that $m_1' = m_2' = m_3' = \checkmark$. By the definition of $\triangleleft$ we find that $m_1 = m_2 = m_3 = \checkmark$, then by applying the induction hypothesis on t_1 with the first premise of both [App] derivations, we have that $\lambda x.\ t_3 \equiv \lambda x'.\ t_3'$. Similarly with induction on t_2 we have that $v_1 \equiv v_1'$. Subsequently, we have $t_3[v_1/x] \equiv t_3'[v_1'/x]$, so by the third premises of both derivations, we have $v_2 \equiv v_2'$.

$\square$

Chapter 3

Contract Inference

In this chapter we investigate some earlier work on propagating contracts by adapting a type inference algorithm. We formalize the problem statement and show that such an approach is not correct.

Suppose we have a function definition that does not implement its specification-contract, and we have proof for this in the form of a valid input that causes an invalid output. We will call this situation the *incorrect implementation scenario*. How could we localize the programming mistake in the definition of this function? A natural idea would be to manually annotate parts of the function's definition with some more contracts and see which parts break and which don't when run with the problematic input. Not only can we narrow down the location of the problem, but by using contracts we write down the intent or specification so we can get a clue *why* the program is incorrect. Since contracts look very similar to types as we know them from the Hindley-Milner types, could we piggyback on the well-known algorithm $\mathcal{W}$ to free the programmer from this annotation burden? Of course, such an algorithm would be of great help in the Ask-Elle programming tutor to provide feedback for students.

Previous effort has been made on this specific idea, in their master theses [25] [16] , Stutterheim and Lauwers introduced a type-system based on Hindley-Milner and presented an accompanying inference algorithm. The basic idea is derived from an example of insertion sort:

```
isort xs =
    let insert = \x ys . case ys of
            [] -> [x]
            (y:ys) -> if x < y
                          then x:y:ys
                          else insert x ys
    in foldr insert [] xs
```

The programming mistake being that in the else branch, the value y is forgotten to be added to the list. The Haskell type of foldr that we are used to is $(a \to b \to b) \to b \to [a] \to b$. Now if we know that the specification contract of `isort` is $[Int] \to \{\texttt{r} : [Int] \mid \texttt{nonDescending r}\}$, we could instantiate b with $\{\texttt{ys} : [Int] \mid \texttt{nonDescending}\}$ and thus derive that `insert` has type $Int \to \{[Int] \mid \texttt{nonDescending}\} \to \{[Int] \mid \texttt{nonDescending}\}$. This contract can then be monitored on uses of `insert`, and we get a more precise idea of where the mistake in the program is.

But their work is not yet finished. For example, dependent contracts are not supported, while these are necessary to describe specifications that relate input to output, like $xs : [a] \to \{\texttt{ys} : [a] \mid \texttt{reverse ys == xs}\}$. There are also some issues remaining regarding subtyping and there has not yet been an actual integration in Ask-Elle. It seems worthwhile to continue their investigations. Furthermore, the idea of trying to infer contracts is not new, [24] presents an algorithm for a limited form of refinement types, using logical qualifiers called liquid types (which stands for logically qualified types). There has also been work in the area of imperative programming languages,

such as [4] on inferring contracts for extracted Java methods from automated refactoring. We will discuss this related work later in Chapter 5.

Algorithm $\mathcal{W}$ is directly connected to the Hindley-Milner type system, so to understand algorithm $\mathcal{W}$ we will take a closer look at the Hindley-Milner type system. In this chapter, we will try to extend the Hindley-Milner system as well as algorithm $\mathcal{W}$ in a step-by-step fashion. We investigate what the differences between contracts and the Hindley-Milner types entail and what difficulties arise when trying to design an inference algorithm similar to algorithm $\mathcal{W}$.

3.1 The Hindley-Milner type system

The Hindley-Milner system [5] is one of the most well-known type systems for the lambda calculus. It consists of a set of deduction rules to prove type judgements. A type judgement is of the form

$$\Gamma \vdash t : \sigma$$

It specifies a ternary relation between a context Γ, a lambda term t and a typescheme σ. We read this as "In the context Γ term t has type(scheme) σ".

Some terms can have multiple types in the same context. For example, take this derivation of a type for the identity function $\Gamma \vdash \lambda x.x : \text{Bool} \rightarrow \text{Bool}$

$$\cfrac{\cfrac{\cfrac{\cfrac{x : \alpha \in [x : \alpha]}{[x : \alpha] \vdash x : \alpha}\,[\text{Var}]}{\epsilon \vdash \lambda x.x : \alpha \rightarrow \alpha}\,[\text{Abs}]}{\epsilon \vdash \lambda x.x : \forall \alpha.\alpha \rightarrow \alpha}\,[\text{Gen}]}{\epsilon \vdash \lambda x.x : \text{Bool} \rightarrow \text{Bool}}\,[\text{Inst}]$$

Note that the last two judgements are both valid typings for the identity and we could write a similar derivation for $\Gamma \vdash \lambda x.x : Int \rightarrow Int$. In the Hindley-Milner system, it is guaranteed that any typable term has a *principal type*: a most general type of which all other types are a specialization [5]. In the above case, $\forall a.a \rightarrow a$ is the principal type.

Since the deduction rules can be stated in a syntax-directed manner, it is possible to have an algorithm that can test whether a type judgement is valid, also known as a static type-checker.

3.2 User defined contracts or refinements

In our first attempt to modify the Hindley Milner system, we will look at *user defined contracts*. Any contract library in Haskell allows the programmer to express a certain predicate on a term. For example, using the syntax of `typed-contracts` [13], we can write:

```
let pos = Prop (\x -> x > 0)
    eq1 = Prop (\x -> x == 1)
in Function pos (\_ -> eq1)
```

This defines a contract for a function that accepts a positive integer and returns an integer that equals 1. The `Prop` constructor has type $(a \rightarrow Bool) \rightarrow Contract\ a$, giving the programmer the possibility to write down a predicate for a value. The `Function` constructor has type $Contract\ a \rightarrow (a \rightarrow Contract\ b) \rightarrow Contract\ (a \rightarrow b)$.

To fit this in the Hindley Milner system, we extend the grammar of types:

$$\begin{aligned} \tau &::= \tau \rightarrow \tau \mid \alpha \mid \{\mathtt{x} : \tau_{base} \mid t\} \\ \tau_{base} &::= \text{Bool} \mid \text{Int} \mid \ \ldots \end{aligned}$$

Any base type is now refined by a predicate term t. Such types are often called refinement types in literature [8] so we will call them the same. We will now summarise the consequences of this extension for the type system, inference algorithm, unification algorithm and correctness of inference.

Consequences for the Hindley-Milner type system

In what ways does this affect the type system, and more importantly, algorithm $\mathcal{W}$? For the type system, it means we want to allow proofs for judgements like $\epsilon \vdash 5 : \{\mathtt{x} : Int \mid \mathtt{x <= 10}\}$. In this particular example we could just prove it by falling back on a semantics of our language:

$$\frac{\mathtt{5\ <=\ 10} \Downarrow_{\checkmark} \mathtt{True}}{\epsilon \vdash \mathtt{5} : \{\mathtt{x} : Int \mid \mathtt{x\ <=\ 10}\}}$$

Effectively "typechecking by evaluation". So typechecking this term is still possible, although termination is not guaranteed as it requires evaluation of the refinement term under consideration.

But this idea does not extend to functions. This is where it gets more complicated, because for a function type $\{\mathtt{x} : \tau_1 \mid \varphi\} \to \{\mathtt{y} : \tau_2 \mid \psi\}$ we get by the [Abs] rule that

$$\frac{[x : \{\mathtt{x} : \tau_1 \mid \varphi\}] \vdash t : \{\mathtt{y} : \tau_2 \mid \psi\} : \tau}{\epsilon \vdash \lambda x.t : \{\mathtt{x} : \tau_1 \mid \varphi\} \to \{\mathtt{y} : \tau_2 \mid \psi\}}$$

For closed terms we could use the evaluation semantics of the language, but in this case t is not closed. In order to evaluate open terms, we need some sort of environment with expressions bound to all free variables. But all we know about the free variable x is that φ holds. We could formulate the problem as $\forall \mathtt{x} : \tau_1.\varphi \Rightarrow \psi \Downarrow_{\checkmark} True$, but since there are infinitely many values of type Int for x, typechecking by evaluation is clearly not an option anymore.

Even worse, the seemingly small addition of refinements allows to state complex theorems:

```
f x = if x == 1
         then 1
         else if even x
            then f (div x 2)
            else f (3*x + 1)
```

f takes a number x and unless that number equals 1, it recursively continues with $\frac{x}{2}$ when x is even or $3x + 1$ when x is odd. Suppose we want to type check $\epsilon \vdash \mathtt{f} : \{\mathtt{x} : Int \mid \mathtt{x>0}\} \to \{\mathtt{y} : Int \mid \mathtt{y==1}\}$. This exact problem is known as the Collatz conjecture and is still an unsolved problem in mathematics [2]. Clearly, the expressiveness of refinement types comes at the cost of undecidability of typechecking.

Still, much work has been done on static checking of refinement types [24]. The common way to obtain a terminating algorithm is by limiting the language that can be used for the refinements to that of a decidable logic, and then using a automated theorem prover to solve a set of constraints obtained by the refinement types. For example [26] use QF-EUFLIA, which is a decidable logic of equality, uninterpreted functions and linear arithmetic. In another direction there has been work on a hybrid approach: doing as many checks statically and deferring the rest to run-time monitoring [10]. In our case, it is not acceptable to limit the language of refinements since we need to express the complete specification of a programming exercises.

Consequences for correctness of inference

In the previous section, we found that refinement types come at a cost: we lose the ability to check a type at compile time. This seems highly problematic for any inference algorithm then, since type inference is supposed to produce valid typings. This leads us to an important realisation: it is not our intended goal to infer valid types or contracts: in our "incorrect implementation scenario"

we already know that the program does not meet its specification (its type) and have a counter example to prove it. But if we do not want to infer valid typings, we have to infer invalid typings. But which ones exactly? We will try to give a formal criterion in section 3.3.

So while algorithm $\mathcal{W}$'s soundness and completeness are defined with respect to the type system rules, any correctness properties for the adapted inference algorithm should be defined in terms of the semantics of contract monitoring.

Consequences for the inference algorithm

Suppose we adapt algorithm $\mathcal{W}$ and treat refined types just like base types are treated in Hindley-Milner. Some questions arise that we try to discuss here. A first obvious question would be: how would refinement types enter during the inference process? When inspecting the rules of ordinary Hindley-Milner, we find that the primitive types like *Int* and *Bool* enter because they appear in the rules for constants and primitive operators like `+`. For refinement types, we cannot do the same, as there are many equally valid refinement types. Take for example the plus operator:

$$
\begin{aligned}
\texttt{(+)} \;\texttt{::}\;\; & \{\texttt{x} : \mathit{Int} \mid \texttt{True}\} \to \{\texttt{y} : \mathit{Int} \mid \texttt{True}\} \to \{\texttt{z} : \mathit{Int} \mid \texttt{True}\} \\
\texttt{(+)} \;\texttt{::}\;\; & \{\texttt{x} : \mathit{Int} \mid \texttt{x > 0}\} \to \{\texttt{y} : \mathit{Int} \mid \texttt{y > 0}\} \to \{\texttt{z} : \mathit{Int} \mid \texttt{z /= 1}\} \\
\texttt{(+)} \;\texttt{::}\;\; & \{\texttt{x} : \mathit{Int} \mid \texttt{even x}\} \to \{\texttt{y} : \mathit{Int} \mid \texttt{odd x}\} \to \{\texttt{z} : \mathit{Int} \mid \texttt{odd z}\} \\
\ldots &
\end{aligned}
$$

One could come up with arbitrary many valid types for the `+` operator. Which type should be inferred? Although in regular Hindley-Milner terms can have an arbitrary amount of types as well, such as *id*, we always infer the principal type, which in a sense "captures" all other possible types: any other type is a specialization of the principal type and the process of unification constructively shows this. With refinement types, we do not have a similar concept to the principal type. Although such a property is nice to have, it does not have to be fatal for an adapted algorithm $\mathcal{W}$, we might be able to live with just any relevant inferred type. The question that remains however, is where those relevant refinements should come from.

In the "incorrect implementation scenario", we of course have the top-level contract at our disposal. It gives us refinements of the top-level argument types and a refinement for the result type. In his thesis, Stutterheim notes that "In essence, [usage of the top-level contract by the inference algorithm] is not very different from the way an explicit type annotation is persisted through a program in type inference". In the classic Hindley-Milner system, it is not difficult to add a rule for type annotations:

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash (t :: \tau) : \tau}$$

That is, an annotated term $t :: \tau$ has type τ if we can derive that exact type for the term. Also, for algorithm $\mathcal{W}$ this would translate to another case of unification:

$$
\begin{aligned}
&\mathcal{W}(\Gamma, t :: \tau_1) = \\
&\qquad \mathit{let}\ (\tau_2, \theta_1) = \mathcal{W}(\Gamma, t) \\
&\qquad\quad\ \ \theta_2 = \mathcal{U}(\tau_1, \tau_2) \\
&\qquad \mathit{in}\ (\theta_2\tau_2, \theta_2 \circ \theta_1)
\end{aligned}
$$

So there seems to be a way for refinements to enter during the inference: by means of annotations. But a more important question follows: how do these types propagate through the program and does this happen correctly? The core mechanism that handles this in algorithm $\mathcal{W}$ is the unification subroutine $\mathcal{U}$ which we will discuss in the next subsection.

Another question is, can we actually infer new refinement types from existing ones. For example, if we consider

$$(+) \ :: \quad \{\texttt{x} : Int \mid \texttt{x > 0}\} \to \{\texttt{y} : Int \mid \texttt{y > 0}\} \to \{\texttt{z} : Int \mid \texttt{?}\}$$

Can we infer a refinement that holds on the result type? One correct refinement is $True$, but this does not tell anything new. In this case, we could do some integer specific reasoning to deduce that $\{\texttt{z} : Int \mid \texttt{z > 0}\}$. Unfortunately, these kind of properties cannot be computed in general because of Rice's theorem. But remember, we are not trying to infer correct types per se, just types that follow from the top-level contract.

Consequences for unification

Two types τ_1 and τ_2 are unifiable if there exists a substitution θ such that $\theta\tau_1 = \theta\tau_2$. θ is called a *unifier* of τ_1 and τ_2. In algorithm $\mathcal{W}$, Robinson's unification algorithm [23] is used to compute a unifier, denoted as a partial function $\mathcal{U} : \tau \times \tau \rightharpoonup Subst$.

Now, in the presence of refinements, what should we think of

$$\mathcal{U}(\{\texttt{x} : Int \mid \texttt{even x}\}, \{\texttt{y} : Int \mid \texttt{odd (y+1)}\})$$

There exists no unifier for these types, but yet they are clearly equal! Equality of arbitrary refinements is undecidable, so that seems problematic. And what about

$$\mathcal{U}(\{\texttt{x} : Int \mid \texttt{even x}\}, \{\texttt{y} : Int \mid \texttt{y > 10}\})$$

These types differ, and if we treat refinement types like other base types, the algorithm would stop and give a type mismatch error. But this is not the behaviour we seek! We are fully aware that the program under consideration might contain refinement type errors and more importantly, it could well be the case that these types are not yet final, just like a variable can instantiated, a refinement might get even more refined. Suppose that this specific example of unification is caused by a function application `f x`. We could argue that this is evidence that the types should be even stronger, so the "intersection" $\{\texttt{x} : Int \mid \texttt{even x \&\& x > 10}\}$ might be a desired outcome. After all, the point of unification in algorithm $\mathcal{W}$ is to produce substitutions that find the most "specialized" outcome. Now suppose that we have a way of recording the specialization of this type (ordinary substitutions on variables do not suffice to state this) there are issues regarding subtyping. Consider for instance the case where we have an application `square x` and we know

$$\begin{aligned}\Gamma = \{&x \mapsto \{\texttt{x} : Int \mid \texttt{even x}\} \\ &square \mapsto \{\texttt{x} : Int \mid \texttt{True}\} \to \{\texttt{y} : Int \mid \texttt{True}\}\}\end{aligned}$$

Then from unification of the argument type with the parameter type, we learn that $\texttt{x} : \{\texttt{x} : Int \mid \texttt{even x \&\& True}\}$ and $square : \{\texttt{x} : Int \mid \texttt{even x \&\& True}\} \to \{\texttt{y} : Int \mid \texttt{True}\}$. This last type is not valid in general, not every call to *square* should enforce its argument to be even. The resulting refinement is *stronger* than (i.e. a subtype of) both types being unified.

But why does this problem not occur in ordinary Hindley-Milner? In the case of `id 5` we unify $\mathcal{U}(\alpha_n, Int)$ to specialize id to $Int \to Int$, but obviously not every call to id is of type $Int \to Int$. It is the quantifiers of type schemes that "protect" other occurrences of id. More specifically, we have that $\Gamma(x) = Int$ and $\Gamma(id) = \forall\alpha.\alpha \to \alpha$, and algorithm $\mathcal{W}$ instantiates *this occurrence* of id with a fresh variable α_n for some unique number n.

So one might argue that this idea could be extended to a type system with refinement types: have every function application `f x` only compute the intersection of refinements for that specific occurrence of `f`. The problem with this idea is that no more refinement types are distributed through the program, all information is kept "local" at the application. Whereas vanilla HM also allowed substitutions to have a "global effect", since inference variables not only originate

from instantiating polymorphic types, but are also introduced at binding sites, such as a lambda abstraction.

Stutterheim makes a different choice for unification, by not taking the intersection, but the strongest of the two types, with the idea that if these types are to "match", one of them should be a subtype of the other. He approaches this by having a flattened type structure: there is no distinction between base types, variables or refinements. Instead there are only predicates to describe all these basic types. E.g. type variables are now represented as `true`, Int as `isInteger x` etc. Furthermore he indexes every such predicate type[1] with unique integers so that substitutions prescribe which exact contract to substitute for another. This labeling has the effect that distinct variables can now be represented (e.g. $true_0$ and $true_1$), but also to specify that specific refinements might be even stronger ($isInteger_5$ might be found to be $isEven_0$). His adapted unification algorithm then states that

$$\begin{aligned}\mathcal{U}(c_n, c_m) &= [c_n \mapsto c_m] \quad \text{iff } c_n \notin free(c_m) \wedge [\![c_m]\!] \subseteq [\![c_n]\!] \\ \mathcal{U}(c_n, c_m) &= [c_m \mapsto c_n] \quad \text{iff } c_m \notin free(c_n) \wedge [\![c_n]\!] \subseteq [\![c_m]\!]\end{aligned}$$

Where n, m are indices for the predicate types. $[\![c_n]\!] \subseteq [\![c_m]\!]$ denotes a subtyping relation. One of the problems with this approach is that deciding such a subtype relation is not possible for arbitrary contracts, and Stutterheim gives no hints on how to compute this. Furthermore, we have that contravariant positions in a function type are not handled correctly with respect to the subtyping relation. On a different angle, the indexing does not completely help with solving the above problem that occurs with `square x`: algorithm $\mathcal{CW}$ finds that $\texttt{square} : \texttt{true}_1 \rightarrow \texttt{true}_2$ and when applied to $\texttt{x} : \texttt{even}_3$ we get from unification that $[\texttt{true}_1 \mapsto \texttt{even}_3]$. Note that this is better than not having any indices, since the substitution states only to replace $\texttt{true}_1$, so it will not affect any other $\texttt{true}_k$ with $k \neq 1$. It still causes any other application of `square` to incorrectly have a stronger precondition.

Lauwers also identifies this problem, but proposes a unification algorithm that does not compare subtyping of predicate types and fails unification completely if such types differ, meaning that the algorithm does not produce any meaningful result when the types get interesting.

3.3 Correctness properties for an inference algorithm

There are two important properties of algorithm $\mathcal{W}$:

- **Soundness** When $W(\Gamma, t) = \sigma$ then $\Gamma \vdash t : \sigma$
- **Completeness** When t is typable in Γ (there exists a σ_1 such that $\Gamma \vdash t : \sigma_1$) then $W(\Gamma, t) = \sigma$, where σ is the principle type of t.

Soundness ensures that W only produces correct typings, while **Completeness** ensures that it can do so for every typable term (and additionally producing the principal type).

In chapter two, we have seen that contract monitoring is a run-time process, checks are performed for specific program executions only. So while monitoring is generally not capable of showing that a term satisfies its refinement type, it *can* show that a term does not satisfy its refinement type, i.e. $\Gamma \nvdash t : \sigma$.

Definition 3.3.1. An annotator is a mapping $A : t \times \sigma \rightarrow t^+$ which maps a term and top-level contract to a term, such that $\textbf{forget}(A(t, \sigma)) \equiv t$.

Intuitively, an annotator can only modify the program by adding **monitor** constructs and constructs that do not "change" the semantics of the original program. The trivial annotator A_0 is defined as $A_0(x, c) = \textbf{monitor}\ c\ x$

For annotators we use the following shorthand notation:

[1] Stutterheim refers to these as user-defined contracts, but in this setting we try to avoid the word contract

- $A(t, \sigma)\checkmark$ means $\forall \overrightarrow{x}.\ A(t, \sigma)\,\overrightarrow{x}\checkmark$

Definition 3.3.2. An annotator A is *consistent* when for all terms t, top-level contracts σ

1. $A_0(t, \sigma)\checkmark \Rightarrow A(t, \sigma)\checkmark$
2. $\forall \overrightarrow{x}.\ A_0(t, \sigma)\,\overrightarrow{x}\,\lightning \Rightarrow A(t, \sigma)\,\overrightarrow{x}\,\lightning$

In other words, correct programs should not raise violations and violations raised by A_0 are preserved by A. The exact violations might differ though and A might give violations in cases that A_0 does not.

3.4 Refinement contracts and strengthening

In the thesis of Stutterheim [25], the algorithm $\mathcal{CW}$ is presented, which resembles algorithm $\mathcal{W}$. It computes a set of substitutions and a contract for the complete term. Just like algorithm $\mathcal{W}$, we can use the substitutions to obtain a contract for every subexpression. The differences include changes to the unification algorithm to handle subtyping and addition of realistic language constructs to the underlying lambda-calculus.

Another important change to algorithm $\mathcal{W}$ is to index all contracts with an integer, this is necessary since subtyping might cause certain contracts to strengthen. E.g. when we have to compute the following unification $U(int_n, nat_m)$, the produced substitution $(int_n \mapsto nat_m)$ without indices would affect any *int* in the program to be substituted for *nat*, which is of course unwanted.

Suppose we are given the following program $t =$

```
\n -> let inc x = x + 1
      in inc (inc n)
```

and a top-level contract $\sigma = \{\texttt{x} : Int \mid \texttt{even x}\} \rightarrow \{\texttt{y} : Int \mid \texttt{even y}\}$. Essentially, algorithm $\mathcal{CW}$ will find that $\texttt{inc} : \{\texttt{x} : Int \mid \alpha\} \rightarrow \{\texttt{y} : Int \mid \beta\}$. And with the knowledge that the function returns a $\{\texttt{x} : Int \mid \texttt{even x}\}$, we will obtain the substitution $\beta \mapsto \texttt{even x}$. However, when annotating the program this causes both occurrences of `inc` to monitor that their output is even, which is not the case for the inner call.

When viewed as an annotator A_{CW}, we have that

$$A_{CW}(t, \sigma)\ \texttt{0} \Downarrow_{\lightning\ \text{“0 + 1 is not even”}} 2 \qquad \text{while } A_0(t, \sigma)\checkmark$$

Thus breaking the first part of the consistency definition. The problem is that `inc` can have a different refined type at every occurrence.

3.5 Type variables and contracts

Lauwers proposes in his thesis a two-step algorithm for inferring contracts [16]. The first step uses algorithm $\mathcal{W}$ to find the usual Hindley-Milner types for all bound identifiers, which are then used as a template for so-called initial contracts. The second step consists of an algorithm somewhat similar to algorithm $\mathcal{W}$, named algorithm $\mathcal{CHW}$. It produces a sequence of substitutions obtained from unification.

The initial contracts from step 1 are constructed in the following way from a Hindley-Milner type

1. A type variable α is directly converted to a contract variable α
2. Every *occurrence* of a type constructor b is converted to a unique (fresh) contract variable

3. Every function type $\tau_1 \to \tau_2$ is converted to a contract $c_1 \to c_2$ by recursively converting τ_1 and τ_2 to c_1 and c_2 respectively

For example, the type of map $(\alpha \to \beta) \to [\alpha] \to [\beta]$ is converted to $(\alpha \to \beta) \to \gamma\alpha \to \delta\beta$. Note the two different contract variables for the list occurrences. It is clear that such contracts (or refinements) do not have to be equal per se. Take for example *tail* :: $[a] \to [a]$, which drops the first element from a list. We know that refinements over the list structure (such as length) are not preserved by this function.

Having equal contract variables is crucial for the rest of the inference process, since they provide the mechanism for "distributing" interesting contract information over the rest of the program. In this section, we argue that even type variables as produced by Hindley-Milner cannot always be used as equal contract (refinement) variables.

Take for example a common Haskell function `until`

$$\texttt{until} :: \forall \alpha.(\alpha \to \mathit{Bool}) \to (\alpha \to \alpha) \to \alpha$$

Suppose now that we have a trivial exercise to write a program that produces an even integer. We use until for a rather artificial implementation:

```
n = until (>= 10) (+1) (-5)
```

For the identifier `until`, the following initial contract is produced: $(\alpha \to \beta) \to (\alpha \to \alpha) \to \alpha$. Subsequently, the second step will unify $U(\alpha, \mathit{even})$, resulting in substitutions that when applied to `until`'s initial contract results in $(\mathit{even} \to \beta) \to (\mathit{even} \to \mathit{even}) \to \mathit{even}$. It should be clear that any application of the second argument `(+1)` will now violate its contract $\mathit{even} \to \mathit{even}$, which should not have been inferred in the first place. In terms of annotators: if we define an annotator $A_{\mathcal{CHW}}$ which annotates the program with contracts as inferred by algorithm $\mathcal{CHW}$, we get

$$A_{CHW}(n, \mathit{even}) \Downarrow_{\lightning:\text{“The argument -5 to the function } (\geq 10) \text{ is not even”}} 2$$
$$\text{while } A_0(n, \mathit{even}) \Downarrow_{\checkmark} 2$$

This breaks the second part of the consistency definition. Clearly there is something wrong with the assumption that equal type parameters from Hindley-Milner carry over to equal type parameters in the setting of refinement types. But why is this? After all, we read a polymorphic type signature as valid *for all types* α *...*

The core of the problem is the [*App*] rule which, in Hindley-Milner, forces the argument type and the domain type to be equal:

$$\frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1 t_2 : \tau_2} \text{[App]}$$

In algorithm $\mathcal{W}$, this rule translates to computing a unification of the two types, resulting in multiple occurrences of the same type variable. With refinement types this rule is too strong: functions tend to be *subtype polymorphic*: any argument is accepted that is a subtype of the domain of the function. For example, a function f that accepts any integer, can also accept an integer that has a refinement type $\{\texttt{x} : \mathit{Int} \mid \texttt{even x} \wedge \texttt{x > 0}\}$. This does not mean that they should be forced to be equal, unification is not the right mechanism to use.

To understand why this subtype polymorphism is incorrectly handled in `until`, consider the definition

```
let until p f x = if p x then x else until p f (f x) in ...
```

The recursive occurrence of `until` may not be polymorphic because the [Let] rule forces the right hand side to use a monomorphic version, as polymorphic recursion is undecidable. Hence

from applying `until p f` on `f x`, a substitution is obtained for `f` that unifies with the unification variable introduced for `x` in the [Abs] rule.

But one could argue that Hindley-Milner types also exhibit some form of subtyping, why is there no similar problem then? Typeschemes have a basic order defined on them, which can be thought of as subtyping. The relation can be defined as follows [2]

$$\frac{\tau' = [\alpha_i := \tau_i]\tau \qquad \beta_i \notin free(\forall\alpha_1 \ldots \forall\alpha_n.\tau)}{\forall\alpha_n.\tau \sqsubseteq \forall\beta_1 \ldots \forall\beta_m.\tau'}$$

For example, $\forall a.a \rightarrow a \sqsubseteq Int \rightarrow Int$. The set of functions with the former type is a subset of the function with the latter type. Or when seen from the perspective of Liskov's substitution principle [17], a value of type $\forall a.a \rightarrow a$ can be safely used in a context where a function of type $Int \rightarrow Int$ is expected, simply by means of instantiation. But note that only typeschemes form such a relation, not monotypes. And since it is syntactically not possible to have typeschemes on contravariant positions, like the domain of a function arrow, functions can not be subtype-polymorphic in their arguments. Any actual application consists of two monotypes.

3.6 Summary

The main reason that we cannot simply adjust algorithm W for propagating (invalid) refinement types over the program is that the essential mechanism for spreading information, namely type variables and substitutions, breaks in the setting of refinement types. Equal type variables are not equal refinement types per se. Furthermore, from a function application the algorithm can no longer conclude that the function's domain is equal to the refinement that might hold for the argument. Circumventing this with universal quantifiers as in polymorphic types defeats the purpose of propagating the refinement types.

[2] from `https://en.wikipedia.org/wiki/Hindley%E2%80%93Milner_type_system#Polymorphic_type_order`

Chapter 4

Contract Propagation and Folds

In this chapter we investigate the possibilities of propagating contracts in a somewhat constrained setting. We drop the idea of using a type inference algorithm and investigate the behaviour of *folds*. First we reason about the use of folds and their relation to inductive properties. Then we describe a simple annotator that propagates contracts over a program defined as a fold. We allow the use of dependent contracts and show that we have defined a consistent annotator.

4.1 Folds

A fold or catamorphism is a recursion scheme for writing recursive programs over a specific datatype. The programmer only writes down the interesting parts of the computation and the fold takes care of the actual recursion. A very common example is the Haskell function `foldr` for lists.

```
foldr :: (a -> b -> b) -> b -> [a] -> b

sum xs = foldr (+) 0 xs
length xs = foldr (\_ x -> x + 1) 0 xs
```

`foldr`'s first two arguments are known as the *algebra* of the fold: for both list constructors, the cons and empty list, the programmer provides a way of computing the final value for which they may use the recursive result of folding the tail.

There also exist folds for other recursive datatypes. In fact, for any datatype there exists a fold that, given an algebra, recursively computes a value. Folds come from category theory, where they are formally defined as the unique homomorphism from an initial algebra into some other algebra in the category of F-algebra's for some endofunctor F. In their classic paper [18], Meijer et al have given a wonderful exposition of how this formal definition of folds (among other recursion schemes) can be applied in functional programming.

Folds are interesting for our problem because they constrain computations to the structural recursion on a datatype, which strongly resembles inductive reasoning of properties over such data. We start by investigating the relation between properties that hold for the fold and for their algebra. For now, we will restrict ourselves to the case of folds over lists.

4.2 Preserving properties

The common way to prove properties over recursive datatypes is by case analysis for the possible constructors and of course, induction for recursive positions. For example, suppose we want to prove a property $\forall xs.\varphi(xs)$ where xs is a list, we can do so by proving the property for the empty list and for the non-empty list by using induction. In order to prove a property of the form

$\forall xs.\varphi(foldr\ f\ e\ xs)$, where f and e form the algebra, we find that some additional properties are required for f and e.

Let $\Gamma = \{\forall x\ r.\varphi(r) \Rightarrow \varphi(f\ x\ r), \varphi(e)\}$ i.e. the assumptions that f preserves φ and the constant e satisfies φ. In natural deduction style, we will now prove:

$$\Gamma \vdash \forall xs.\varphi(foldr\ f\ e\ xs) \tag{4.1}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, \mathrm{IH} \vdash \forall x\ r.\varphi(r) \Rightarrow \varphi(f\ x\ r)}{\Gamma, \mathrm{IH} \vdash \forall x.\varphi(foldr\ f\ e\ xs) \Rightarrow \varphi(f\ x\ (foldr\ f\ e\ xs))}\{r := foldr\ f\ e\ xs\}}{\Gamma, \mathrm{IH} \vdash \forall x\ xs.\varphi(foldr\ f\ e\ xs) \Rightarrow \varphi(f\ x\ (foldr\ f\ e\ xs))} \quad \dfrac{\Gamma, \mathrm{IH} \vdash \mathrm{IH}}{\Gamma, \mathrm{IH} \vdash \forall xs.\varphi(foldr\ f\ e\ xs)}}{\Gamma, \mathrm{IH} \vdash \forall x\ xs.\varphi(f\ x\ (foldr\ f\ e\ xs)}}{\Gamma, \mathrm{IH} \vdash \forall x\ xs.\varphi(foldr\ f\ e\ (x : xs))}\{\text{def. foldr}\}$$

Figure 4.1: Cons case

We then split the proof in two: one for each of the two construtors of the list datatype. In figure 1 we use induction to prove that for a list of the form $x : xs$ the property φ holds. In figure 2, we have a trivial proof for the case the list is empty.

$$\dfrac{\Gamma \vdash \varphi(e)}{\Gamma \vdash \varphi(foldr\ f\ e\ [])}$$

Figure 4.2: Nil case

From these two proofs we can conclude that

$$\dfrac{\dfrac{\text{Cons case} \qquad \text{Nil case}}{\Gamma \vdash \forall xs.\varphi(foldr\ f\ e\ xs)}}{\bigwedge \Gamma \Rightarrow \forall xs.\varphi(foldr\ f\ e\ xs)}$$

We can read $\bigwedge \Gamma$ as stating that the algebra (f, e) is φ-preserving. It is not hard to see that this proof be generalized to arbitrary datatypes. A bit more informal, we have that *if an algebra is φ-preserving, the fold on any list has property φ.* Its contraposition sounds interesting with regard to contract propagation: *if there is a list for which the fold does not have φ, then the algebra is not φ-preserving.*

Back to contracts then: can we, by this reasoning, blame the algebra for not preserving the desired property? Unfortunately, no. We do not exclude a situation in which f is not φ-preserving but φ holds on the fold anyways. Monitoring the algebra would be a mistake. If however, we can prove the reverse implication:

$$\neg \bigwedge \Gamma \Rightarrow \neg\varphi(foldr\ f\ e\ xs) \tag{4.2}$$

we effectively state that a violation which occurs during the monitoring of the algebra implies that we would also have a violation on the monitoring of the fold. This is what it means to be a *cause* of the top-level violation and is what we intuitively expect from a consistent annotator.

Unfortunately, this formula does not hold. Consider the following counter example:

```
f _ 0 = 3
f _ n = n * 2
    e = 2
```

$$\varphi = even$$

The precedent holds, as `f` does not preserve φ, but the answer will be $2^{1+\texttt{length xs}}$ for any list xs, an even integer. So it is not safe to monitor the contract $\top \rightarrow even \rightarrow even$ on the definition of `f` as other calls to the function might trigger unwanted violations.

We can however weaken the antecedent of the contraposition to find the following formula

$$\begin{aligned} &\forall xs.\varphi(foldr\ f\ e\ xs) \Rightarrow \\ &\qquad (xs = y : ys \qquad \Rightarrow \varphi(f\ y\ (foldr\ f\ e\ ys))) \\ &\qquad \wedge\ (xs = [] \qquad \Rightarrow \varphi(e)) \end{aligned} \tag{3}$$

The validity of this formula simply follows from the definition of foldr, and it states that both parts of the algebra, *as used in the fold*, should return a value that has φ.

So from (2), which does not hold, we can conclude that it is not safe to monitor φ on the definition of the algebra, but from (3) that it is justified to monitor φ on the application of the algebra within the fold.

4.2.1 An annotator for `foldr`

The above idea can be turned into a simple annotator:

$$\begin{aligned} A_{foldr}(\texttt{foldr}\ t_1\ t_2, \top \rightarrow \varphi) &= \texttt{foldr}\ (\textbf{monitor}\ (\top \rightarrow \top \rightarrow \varphi)\ \ t_1)(\textbf{monitor}\ \varphi\ \ t_2) \\ A_{foldr}(t, \sigma) &= A_0(t, \sigma) \end{aligned}$$

There are a few things to note about this annotator. First, we observe that the added contracts will be monitored repeatedly on all recursive calls of `foldr`. Furthermore, note that instead of the original $\top \rightarrow \varphi \rightarrow \varphi$ we had in mind, the argument that is the recursive result is *not* monitored by f. This is indeed not necessary, as can be seen with a simple inductive argument: any value recursively computed by the fold is already monitored by the part of the algebra that produced it.

We now show that this annotator is consistent by our definition from Chapter 3. We only show this for the first case of A_{foldr}, as the second is equal to the trivial annotator.

Recall that we use monospace typesetting for variables in λ^+ (`x, y, foldr ...`), and math typesetting for meta variables ranging over terms ($t, x, xs \dots$). Furthermore, $\equiv$ denotes semantic equality as defined in Chapter 2 while $=$ denotes regular meta equality (for example $\textbf{forget}(\texttt{x}) = \texttt{x}$). In the proofs below we use equational reasoning over semantic equality, but since meta equality implies semantic equality we mix the two operators to help the reader in distinguishing them.

Theorem 1. $A_0(\texttt{foldr}\ \ t_1\ t_2, \top \rightarrow \varphi)\checkmark \Rightarrow A_{foldr}(\texttt{foldr}\ \ t_1\ t_2, \top \rightarrow \varphi)\checkmark$

Proof: Assume $A_0(\texttt{foldr}\ \ t_1\ t_2, \top \rightarrow \varphi)\checkmark$. We have to prove $A_{foldr}(\texttt{foldr}\ \ t_1\ t_2, \top \rightarrow \varphi)\checkmark$, meaning $\forall z.\ A_{foldr}(\texttt{foldr}\ t_1\ t_2, \top \rightarrow \varphi)\ z\checkmark$. We do this by induction on z.

case $z = $ `[]`

$$
\begin{aligned}
& A_{foldr}(\texttt{foldr}\ t_1\ t_2, \top \to \varphi)\,\texttt{[]} \\
\{\text{Def. } A_{foldr}\} &= \texttt{foldr}\ (\textbf{monitor}\ (\top \to \top \to \varphi)\ \ t_1)\ (\textbf{monitor}\ \varphi\ \ t_2)\,\texttt{[]} \\
\{\text{Def. } \texttt{foldr}\} &\equiv \textbf{monitor}\ \varphi\ \ t_2 \\
\{\text{Def. } \texttt{foldr}\} &\equiv \textbf{monitor}\ \varphi\ \ (\texttt{foldr}\ t_1\ \ t_2\ \ \texttt{[]}) \\
\{\text{Lemma 2.4.4}\} &\equiv (\textbf{monitor}\ (\top \to \varphi)\ \ \texttt{foldr}\ \ t_1\ \ t_2)\,\texttt{[]} \\
\{\text{Def. } A_0\} &= A_0(\texttt{foldr}\ t_1\ t_2, \top \to \varphi)\,\texttt{[]} \\
& \text{which is violation free by assumption}
\end{aligned}
$$

case $z = x : xs$

$$
\begin{aligned}
& A_{foldr}(\texttt{foldr}\ t_1\ t_2, \top \to \varphi)(x : xs) \\
\{\text{Def. } A_{foldr}\} &= \texttt{foldr}\ (\textbf{monitor}\ (\top \to \top \to \varphi)\ \ t_1)\ (\textbf{monitor}\ \varphi\ \ t_2)\ (x : xs) \\
\{\text{Def. } \texttt{foldr}\} &\equiv (\textbf{monitor}\ (\top \to \top \to \varphi)\ \ t_1)\ x \\
& \qquad (\texttt{foldr}\ (\textbf{monitor}\ (\top \to \top \to \varphi)\ \ t_1)(\textbf{monitor}\ \varphi\ \ t_2)\ xs \\
\{\text{Def. } A_{foldr}\} &= (\textbf{monitor}\ (\top \to \top \to \varphi)\ \ t_1)\ x\ (A_{foldr}(\texttt{foldr}\ t_1\ t_2, \top \to \varphi)\ xs) \\
\{\text{IH \& Lemma 2.4.5}\} &\equiv (\textbf{monitor}\ (\top \to \top \to \varphi)\ \ t_1)\ x\ (\textbf{forget}(A_{foldr}(\texttt{foldr}\ t_1\ t_2), \top \to \varphi)\ xs) \\
\{\text{def. annotator}\} &\equiv (\textbf{monitor}\ (\top \to \top \to \varphi)\ \ t_1)\ x\ (\texttt{foldr}\ t_1\ t_2\ xs) \\
\{\text{Lemma 2.4.4}\} &\equiv \textbf{monitor}\ \varphi\ \ t_1\ \ x\ \ \texttt{foldr}\ \ t_1\ \ t_2\ \ xs) \\
& \text{which is violation free by assumption}
\end{aligned}
$$

The equality labeled {IH & Lemma 2.4.5} is an important step. The induction hypothesis states that $A_{foldr}(\texttt{foldr f e}, \top \to \varphi)xs\checkmark$, which by Lemma 2.4.5 equals $(A_{foldr}(\texttt{foldr}\ t_1\ t_2, \top \to \varphi)\ xs)$

Theorem 2.
$\forall xs.\ (A_0(\texttt{foldr}\ t_1\ t_2, \top \to \varphi)\ xs\lightning \Rightarrow A_{foldr}(\texttt{foldr}\ t_1\ t_2, \top \to \varphi)\ xs\lightning)$

Proof

case `[]`

$$
\begin{aligned}
& \text{assume } A_0(\texttt{foldr}\ t_1\ t_2, \top \to \varphi)\,\texttt{[]}\lightning \\
& A_{foldr}(\texttt{foldr}\ t_1\ t_2, \top \to \varphi)\,\texttt{[]} \\
\{\text{As in Theorem 1}\} &\equiv A_0(\texttt{foldr}\ t_1\ t_2, \top \to \varphi)\,\texttt{[]} \\
& \text{which raises a contract violation by assumption}
\end{aligned}
$$

case $x : xs$

$$
\begin{aligned}
& \text{assume } A_0(\texttt{foldr}\ t_1\ t_2, \top \to \varphi)(x : xs)\lightning \\
& A_{foldr}(\texttt{foldr}\ t_1\ t_2, \top \to \varphi)(x : xs) \\
\{\text{As in Theorem 1}\} &\equiv (\textbf{monitor}\ (\top \to \top \to \varphi)\ \ t_1)\ x\ (A_{foldr}(\texttt{foldr}\ t_1\ t_2, \top \to \varphi)\ xs) \\
& \text{for readability, let } r = A_{foldr}(\texttt{foldr}\ t_1\ t_2, \top \to \varphi)xs \\
\{\text{Lemma 2.4.4}\} &\equiv \textbf{monitor}\ \varphi\ \ (t_1\ \ x\ \ r) \\
& \text{if } r\lightning \text{ then also } t_1\ x\ r\lightning \text{ by the [App] rule, and subsequently} \\
& \textbf{monitor}\ \varphi\ \ (t_1\ \ x\ \ r)\lightning\text{ , by the [Mon-*] rules.} \\
& \text{otherwise } r\checkmark\text{, by Lemma 2.4.5 } r = \textbf{forget}(r) = \textbf{forget}(A_{foldr}(\texttt{foldr}\ t_1\ t_2, \top \to \varphi)\ xs) \\
& = \texttt{foldr}\ t_1\ t_2\ xs \text{ by definition of annotator, so } \textbf{monitor}\ \varphi\ \ (t_1\ \ x\ \ r)\lightning \\
& \text{by assumption}
\end{aligned}
$$

4.3 Dependent contracts

In this section we will extend the annotator for folds to handle more contracts. Note that A_{foldr} only works for contracts of the form $\top \to \sigma$. While this is useful to write some properties, often it is not enough for a complete specification. For the sorting function we can write

$\top \to \{$`r` $: [Int] \mid$ `nondesc r`$\}$

```
where nondesc [] = True
      nondesc [x] = True
      nondesc (x:y:xs) = x <= y && nondesc (y:xs)
```

However, this is not a full specification of a sorting function, for example `const []` will never give any contract violation.

The complete specification can be expressed with a dependent contract

$xs : \top \to \{$`r` $: [Int] \mid$ `nondesc r && r ‘isPermOf‘ xs`$\}$

```
where nondesc [] = True
      nondesc [x] = True
      nondesc (x:y:xs) = x <= y && nondesc (y:xs)
      isPermOf xs ys = sort xs == sort ys
```

One more advantage of this constrained setting is that, as opposed to the type inference approach, it is quite clear how dependent contracts should be propagated.

$A_{\pi foldr}(\texttt{foldr f e}, (\texttt{xs} : \top) \to \{\texttt{y} : \tau \mid \varphi\}) = \texttt{foldr'}\ (\lambda xs.\varphi)\ \texttt{f e}$
$A_{\pi foldr}(t, \sigma) = \texttt{monitor}\ \sigma\ \texttt{t}$

```
where foldr' c f e xs = monitor (c xs) $ case xs of
                          []     -> e
                          (y:ys) -> f y (foldr' c f e ys)
```

Instead of annotating the algebra we change the definition of `foldr` to monitor the dependent contract with the list being folded over. Note that the term φ might have a free variable xs which is therefore bound using the lambda. The proof for consistency of this annotator can be found in Appendix B.

4.4 On the definition of a consistent annotator

At this point we can better explain the considerations and subtleties for the definition of a consistent annotator in Chapter 3. Originally, we came up with a different definition:

$$\forall x.\ A(t,\sigma)\overrightarrow{x} \lightning \iff A_0(t,\sigma)\overrightarrow{x} \lightning \tag{4.3}$$

This definition came from the intuition that the behaviour of the trivial annotator is always "right": it monitors the specification on top-level, treating the program as a black box. We found however, that A_{foldr} was not consistent by this definition:

$A_0(\texttt{foldr (+) 0}, \top \to even)\texttt{[1,2,3]} \Downarrow 6$
however
$A_{foldr}(\texttt{foldr (+) 0}, \top \to even)\texttt{[1,2,3]} \Downarrow \lightning$: "3 + 0 is not even"

At first sight, the behaviour seems indeed incorrect. But as we inspect the contract, we find that the violation does make sense. The recursive call encounters the list `[3]`, which itself is a counter example for the given contract (which is supposed to hold for all inputs).

Furthermore, we found a very subtle point in the definition of an annotator: the requirement that $\textbf{forget}(A(t, \sigma)) = t$ was intended to be *syntactical equality*. In other words, an annotator should only "decorate" the program with **monitor** functions. The reason to do so was to exclude bogus annotators that produce different programs or artificial constructions that could circumvent the definition of consistency. For example:

$$A_{cheat}(t, \top \to \varphi)x = \texttt{if } \varphi \texttt{ (t x) then t x else } A_{incorrect}(t, \top \to \varphi)x$$

This annotator effectively checks whether the toplevel annotator would succeed, and if so, also succeeds by not annotating the program. This always satisfies the first part of the consistency definition. The second part of the definition can now easily be satisifed by incorrect annotators like $A_{false}(t, \sigma)x = \lightning$: "always fails" or A_{CHW}. It became apparant that this restriction of syntactic equality was too strong: when annotating with dependent contracts we also have to modify the program by adding extra parameters. As a solution, we have decided that we want *semantic equality* after removing the **monitor** constructs, keeping in mind that these extra introduced parameters are exclusively used in the **monitor** function. After the **forget** operation, the program will never use the parameters, thus semantic equality for this annotator is preserved. Furthermore, a contract is no longer a Bool-valued function, but rather an abstract datatype that can only be used by **monitor** and constructed with $\to$ and `Prop` which wraps a Bool-valued function. This excludes annotators such as A_{cheat}.

4.5 Propagation in other calls to foldr

The propagation of contracts by A_{foldr} is limited to programs directly expressed as a fold. A natural question is whether this annotation mechanism extends to other uses of folds in the program. Consider this example

```
p = foldr f e . (1:)
```

And suppose that the specification for p is the contract $\top \to \{\texttt{x} : Int \mid \texttt{isPrime x}\}$. This is the algebra:

```
    where f 1 x = (x-x) + 2
          f n x = n + x
          e = 42
```

Clearly, the program is correct with respect to the specification, as it will always return `2`. The propagation technique of A_{foldr} cannot be used though, as it would cause contract violations in recursive applications of the algebra: not all intermediate fold results are prime numbers. Effectively, this occurrence of `foldr f e` has a restricted domain: the image of `(1:)`, non-empty lists of which the first element is a `1`. So any contract $\top \to \varphi$ that should hold for `p` does not hold for `foldr f e` after composing with a different function that might restrict the domain. A correct contract to use for propagation `foldr f e` with would be $\{\texttt{xs} : [Int] \mid \texttt{take 1 xs == [1]}\} \to \{\texttt{r} : \tau \mid \varphi\}$. Unfortunately, computing the image of a function is undecidable in general.

4.6 Generalizing the annotator

Folds are not exclusive to lists. When using the functor representation of datatypes, we can simply generalize the annotator for foldr:

$$A_{\pi fold}(\texttt{fold alg}, (x : \top) \to \varphi) = \texttt{fold' } \varphi \texttt{ alg xs}$$
$$A_{\pi fold}(t, \sigma) = \textbf{monitor } \sigma \texttt{ t}$$

```
where fold' c alg x =
           (monitor (c x) . alg . fmap (fold' c alg) . out) x
```

Again, we define a slightly modified fold combinator that recursively monitors every usage of the algebra.

We omit a consistency proof as generic datatypes are not really expressible in our language. This example is purely to show how one could generalize fold annotators.

Chapter 5

Related work and future research

In this chapter we discuss some relevant related work and describe possible future research to continue the current effort.

5.1 Liquid types

Liquid types, short for Logically Qualified Data Types, is a system that combines Hindley-Milner type inference with predicate abstraction to automatically infer dependent refinement types to prove safety properties of OCaml programs [24]. Liquid types are a subset of general refinement types in the sense that only certain refinements are allowed. The system considers only predicates that are formed as a conjunction of a fixed set of *qualifiers*. In the paper, the algorithm DSOLVE is introduced, which performs the liquid type inference. This seems very relevant in the context of Chapter 3 as it does seem possible to infer refinement types for programs.

The inference process is started by introducing templates, based on the Hindley-Milner types, refinement variables are introduced for each base type, representing the yet unknown refinements. The second step is to introduce constraints: *well-formedness* constraints that ensure that the refinements can only refer to program variables that are "in scope". *Subtyping constraints* express a dataflow relation. For instance that the true branch is a subtype of the complete if-then-else type (with the assumption that the condition is true) and similarly for the else branch (with the condition assumed to be false). After collecting the constraints, they are solved using a least fixpoint-like algorithm for the well-formedness constraints and an embedding in a decidable logic for the subtyping constraints that is solved by any automated theorem prover.

The problem is that for such a system to work, the predicates have to be embeddable in a decidable logic. This is useful for inferring and checking specific properties. In the paper, we find examples of properties that can be expressed with equality, uninterpreted functions and arithmetic. The work hints at another set of qualifiers to proof safe array bounds checking, but it seems far from obvious that top-level contracts in Ask-Elle which specify complete specifications fit in any decidable logic.

Also note that this is a proper inference system in the sense that inferred refinement types are valid typings (as opposed to the motivation of using algorithm $\mathcal{W}$ for propagating "incorrect" properties). When applying their DSOLVE algorithm on a student program it would find some refinement type that holds for the program. The work mentions manual annotations but does not show how this is incorporated by DSOLVE.

5.2 Contracts and laziness

In Chapter 2, we decided to present a strict semantics for a reason. In "The interaction of contacts and laziness" [7], Degen et al. investigate the semantics of contract monitoring in a lazy language and analyze existing explorations of the design space. They note that there is no agreed-upon

intended meaning or theory. The work compares three different approaches by means of two criteria:

Meaning preservation which informally means that monitoring a contract either results in a contract violation or leaves the behaviour of the program unchanged.

Completeness states that every violation of a contract is detected in the form of an exception (or in our case a violation monitoring result).

Note that meaning preservation is similar to the requirement in our definition of an annotator, i.e. that $\mathbf{forget}(A(t, \sigma)) \equiv t$. The motivation behind meaning preservation is that contract monitoring should not make a program *more strict*. Whereas the annotator requirement is merely meant to exclude invalid program transformations. We could prove that in a strict language, an annotator will always produce meaning preserving programs.

What Degen et al. show is that for a lazy language, it is not possible to have both properties at the same time, although each property can be achieved in isolation. For example, HJL monitoring [13] is not complete as its monitor operation (called `assert`) does not guarantee checks of preconditions:

```
assert :: Contract a -> a -> a
...
assert (Fun c f) a = (\x -> assert (f x) (a x)) . assert c
```

When `f` and `a` are not strict in `x` (which is an expression of the form `assert c y`), `c` will never be monitored.

On the other hand, eager contract monitoring [6] gives up meaning preservation in favor of completeness. It achieves this by adding many `seq` constructs in order to make functions always check precondition arguments regardless of whether the function is lazy or not. Similarly, contracts on pairs are strict on both components. Obviously this could change the semantics of the program.

Now suppose we would adapt the operational semantics of this thesis for a call-by name semantics. The question then is, in the context of Ask-Elle, which of the two properties is preferable and how would this affect the proofs for consistent annotators? As the authors suggest, for programs depending on laziness meaning preservation is more important. On first sight this seems right, after all it seems wrong to change the behaviour of the submitted program and in our formalism consisten annotators are not allowed to do so. As a consequence, we lose completeness which implies that in some cases there would be fewer feedback as some violations may not be raised. It should be investigated whether this could lead to non-sensical feedback of other contracts being violated instead.

5.3 Mechanizing the formalization

All current reasoning about consistent annotators is done in "pen-and-paper" style, with a complementary implementation of the language whereas this seems perfectly suited for a dependently typed language such as Coq or Agda [9, 21]. This could combine the implementation with the proofs and have the proofs machine-checked. This could be an interesting exercise for a student learning a depedently-typed language.

5.4 Improving location information

The current implementation of the language and annotators does not actually take into account line and column numbers for violation messages. Location info is something that is crucial for a student to have. As Lauwers already made a big effort on incorporating this information in the generation of contracts, it might be a useful effort to improve this.

5.5 Taking contract propagating further

As we have demonstrated, folds are suitable for propagating contracts in a consistent way. We believe that there are more possibilities for such propagation. For instance, any structurally recursive call of the top-level program, a case-split or `if then else` construct at the top-level of the program can all use the top-level contract in some way. It is important to keep in mind that these annotators only work for direct top-level occurrences due to function composition: a program `f . g` with contract $c_1 \to c_2$, it is not generally decidable what the domain contract for `f` is. (see Section 4.5). A possible improvement to this problem is the realization that composition with surjections (e.g. `tail, reverse, drop n` etc.) does give the opportunity of computing the domain contract of `f`.

In practise, students new to Haskell often prefer explicit recursion over the use of `foldr`. The question is how we can use the annotator for folds in such programs. One possibility is to try and apply a program transformation to "extract" a fold from the program code and consequently annotate the fold using A_{fold}. Hu and Iwasaki [14] have described an algorithm that derives a hylomorphism, an unfold followed by a fold. Adapting this algorithm for only folds seems definitely possible, but there are some consequences. Any violation now has to be translated back to the original program code, which is not straightforward. Furthermore, the program transformation has to be proven to be semantically equivalent to obtain any of the consistency properties of the annotator. Another way is to define a new annotator which only annotates programs in which it "recognizes" a fold, which in turn requires its own consistency proof.

Program tranformations can only use the syntactic structure of a student program to propagate contract information. Another possible perspective on the problem is to incorporate propagation in the operational semantics. This might give more possibilities to propagate contracts, for example:

$$\frac{\textbf{if } t_1 \textbf{ then } (\textbf{monitor } c\ t_2) \textbf{ else } (\textbf{monitor } c\ t_3) \Downarrow_m v}{\textbf{monitor } c\ (\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3) \Downarrow_m v}\ \text{[Mon-ITE]}$$

A clear disadvantage of such an approach is that in practice this would require special support in a compiler or interpreter.

5.6 Debuggers

A completely different approach to providing feedback is a semi-automatic one: providing a debugger to the student. This has the important advantage that a student can track the counter example and decide which contracts should hold where, even in places where an automatic analysis would be undecidable. For Haskell, the ghci debugger [12] is the most well known, but there has also been specific work on interactive fault localization with debugging [1]. A disadvantage is the well known problem that debugging with lazy semantics is non-intuitive because evaluation might jump from thunk to thunk. Especially for new students not aware of the operational semantics of the language this might be highly confusing. Furthermore, it might be quite a big engineering effort to integrate such a tool in the programming tutor.

[1] `https://wiki.haskell.org/Hoed`

Chapter 6

Conclusion

We have thoroughly investigated previous work on using an adapted algorithm $\mathcal{W}$ for propagating contracts in student programs. In order to show that such an approach is not correct we formalized the notion of an *annotator*: a program transformation of which the resulting program can additionally produce contract violations but is otherwise semantically equal to the original student program.

The formalization is presented with a lambda calculus λ_+ that has some realistic language constructs. We gave a strict operational semantics and defined behaviour equivalence on terms and values. This equivalence relation formed the basis of a set of lemma's and useful theorems for proving the main criterium of valid program transformation: a *consistent* annotator. Such an annotator transforms programs to monitor contracts that can raise violations in a "consistent" way when comparing with a trivial annotation: only monitoring the top-level contract.

We have used the formalization to show that the algorithm $\mathcal{W}$ approach is incorrect and given examples of annotators that we prove to be consistent, also including dependent contracts. We have implemented this language and the consistent annotators.

The formalization has been implemented in Haskell including the complete λ^+ language, its semantics, annotators and some examples.

Appendix A

Consistency of dependent contract annotator

In this appendix we prove that $A_{\pi foldr}$ from Chapter 4 is consistent.

Lemma A.0.1. $A_0(t, (x : \top) \to \varphi) \equiv \lambda x.\mathbf{monitor}\ \varphi\ \ (t\ \ x)$

Proof

$$
\begin{aligned}
& A_0(t, (x : \top) \to \varphi) \\
\{\eta\text{-reduction}\} \equiv\ & A_0((\lambda x.t\ x), (x : \top) \to \varphi) \\
\{\text{Def. } A_0\} =\ & \mathbf{monitor}\ ((x : \top) \to \varphi)\ \ (\lambda x.t\ \ x) \\
\{\text{Eval-[Mon-4]}\} \equiv\ & (\lambda x.\mathbf{case}\ (\mathbf{monitor}\ \top\ \ \mathtt{x})\ \mathbf{of}\ x \to \mathbf{monitor}\ \varphi\ \ (t\ \ x))) \\
\{\text{Eval-[Mon-1]}\} \equiv\ & (\lambda x.\mathbf{case}\ x\ \mathbf{of}\ x \to \mathbf{monitor}\ \varphi\ \ (t\ \ x)) \\
\{\text{Eval-[Case-1]}\} \equiv\ & \lambda x.\mathbf{monitor}\ \varphi\ \ (t\ \ x)
\end{aligned}
$$

Theorem 3. let $t = \mathtt{foldr}\ \ t_1\ t_2$ and $c = (xs : \top) \to \varphi$ for any terms t_1, t_2 and contract φ. Then $A_0(t, c)\checkmark \Rightarrow A_{\pi foldr}(t, c)\checkmark$

Proof: Assume $A_0(t, c)\checkmark$. We have to prove $A_{\pi foldr}(t, c)\checkmark$, meaning $\forall z.\ A_{\pi foldr}(\mathtt{foldr}\ \ t_1\ t_2, (xs : \top) \to \varphi)\ z\checkmark$. We do this by induction on z.

$$
\begin{aligned}
& \mathbf{case}\ z = \mathtt{[]} \\
& A_0(\mathtt{foldr}\ \ t_1\ t_2, (xs : \top) \to \varphi)\,\mathtt{[]} \\
\{\text{Lemma A.0.1}\} \equiv\ & (\lambda xs.\mathbf{monitor}\ \varphi\ \ (\mathtt{foldr}\ \ t_1\ \ t_2\ \ xs))\,\mathtt{[]} \\
\{\text{Eval-[App]}\} \equiv\ & \mathbf{monitor}\ \varphi[\mathtt{[]}/xs]\ \ (\mathtt{foldr}\ \ t_1\ \ t_2\ \ \mathtt{[]}) \\
\{\text{Def. } \mathtt{foldr}\} \equiv\ & \mathbf{monitor}\ \varphi[\mathtt{[]}/xs]\ \ t_2 \\
\{\text{Eval-[App]}\} \equiv\ & \mathbf{monitor}\ \ ((\lambda xs.\varphi)\,\mathtt{[]})\ \ t_2 \\
\{\text{Eval-[Case-1]}\} \equiv\ & \mathbf{monitor}\ \ ((\lambda xs.\varphi)\,\mathtt{[]})\ \ (\mathtt{case\ []\ of\ \{[]\ ->}\ t_2\ \mathtt{;\ ...\}}) \\
\{\text{Def. } \mathtt{foldr'}\} \equiv\ & \mathtt{foldr'}\ (\lambda x.\varphi)\ t_1\ t_2\ \mathtt{[]} \\
\{\text{Def. } A_{\pi foldr}\} =\ & A_{\pi foldr}(\mathtt{foldr}\ \ t_1\ t_2, (xs : \top) \to \varphi)\,\mathtt{[]}
\end{aligned}
$$

case $z =$ `y:ys`

$$
\begin{aligned}
& A_0(\texttt{foldr}\ t_1\ t_2, (xs:\top) \to \varphi)(y:ys) \\
\{\text{Lemma A.0.1}\} \equiv\ & (\lambda xs.\textbf{monitor}\ \varphi\ \ (\texttt{foldr}\ t_1\ t_2\ xs))(y:ys) \\
\{\text{Eval-[App]}\} \equiv\ & \textbf{monitor}\ \varphi[y:ys/xs]\ \ (\texttt{foldr}\ t_1\ t_2\ (\texttt{y:ys})) \\
\{\text{Def. } \texttt{foldr}\} \equiv\ & \textbf{monitor}\ \varphi[y:ys/xs]\ \ (t_2\ \texttt{y}\ (\texttt{foldr}\ t_1\ t_2\ (y:ys))) \\
\{\text{IH \& Lemma 2.4.5}\} \equiv\ & \textbf{monitor}\ \varphi[y:ys/xs]\ \ (t_2\ \texttt{y}\ (A_{\pi foldr}(\texttt{foldr}\ t_1\ t_2, (xs:\top) \to \varphi)(y:ys)) \\
\{\text{Def. } A_{\pi foldr}\} \equiv\ & \textbf{monitor}\ \varphi[y:ys/xs]\ \ (t_2\ \texttt{y}\ (\texttt{foldr'}\ (\lambda xs.\varphi)\ t_1\ t_2)(y:ys)) \\
\{\text{Def. } \texttt{foldr'}\} \equiv\ & \texttt{foldr'}\ (\lambda xs.\varphi)\ t_1\ t_2\ (y:ys) \\
\{\text{Def. } A_{\pi foldr}\} =\ & A_{\pi foldr}(\texttt{foldr}\ t_1\ t_2, (xs:\top) \to \varphi)
\end{aligned}
$$

Theorem 4.
let $t = \texttt{foldr}\ t_1\ t_2$ and $c = (xs:\top) \to \varphi$ for any terms t_1, t_2 and contract φ. Then
$\forall xs.\ (A_0(t,c)\ xs\lightning \Rightarrow A_{\pi foldr}(t,c) xs\lightning)$

Proof

case $z =$ `[]`, suppose $A_0(t,c)\,\texttt{[]}\lightning$, then

$$
\begin{aligned}
& A_{\pi foldr}(t,c)\,\texttt{[]} \\
&= (\texttt{foldr'}(\lambda xs.\varphi)\ t_1\ t_2)\,\texttt{[]} \\
&= \textbf{monitor}\ \varphi[\texttt{[]}/xs]\ \ t_2 \\
&= (\lambda xs.\textbf{monitor}\ \varphi\ \ t_2)\,\texttt{[]} \\
&= A_0(t,c)\,\texttt{[]}
\end{aligned}
$$

which raises a violation by assumption

case $z = y:ys$, suppose $A_0(t,c)(y:ys)\lightning$, then

$$
\begin{aligned}
& A_{\pi foldr}(t,c)(y:ys) \\
&= (\texttt{foldr'}\ (\lambda xs.\varphi)\ t_1\ t_2)(y:ys) \\
&\equiv \textbf{monitor}\ \varphi[y:ys/xs]\ \ (t_1\ \ y\ \ (\texttt{foldr'}\ \ (\lambda xs.\varphi)\ \ t_1\ \ t_2\ \ ys)) \\
&\equiv \textbf{monitor}\ \varphi[y:ys/xs]\ \ (t_1\ \ y\ \ (A_{\pi foldr}(t,c)\ \ ys))
\end{aligned}
$$

Call this last term r.

If $(A_{\pi foldr}(t,c)\ ys)\lightning$, then $r\lightning$ because of strictness of t_1 and **monitor**.

Otherwise, if $(A_{\pi foldr}(t,c)\ ys)\checkmark$, then $(A_{\pi foldr}(t,c)\ ys) \equiv t\ ys$ by Lemma 2.4.5 and $r \equiv A_0(t,c)(y:ys)$, which raises a violation by assumption

With these two theorems, we have proven that $A_{\pi foldr}$ is consistent. □

Appendix B

Implementation details

We have implemented the presented language λ^+ in Haskell[1] together with the annotators and examples of correct and incorrect student programs attempting to implement insertion sort. The annotator $A_{\pi foldr}$ has been implemented and a basic feedback message for contract violations is produced.

The repository contains different modules for syntax, evaluation and sample programs. All modules are self-documented and the `readme.md` file contains instructions for running the code.

[1] `http://github.com/jaccokrijnen/thesis-implementation`

Bibliography

[1] Tristan OR Allwood, Simon Peyton Jones, and Susan Eisenbach. Finding the needle: stack traces for ghc. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 129–140. ACM, 2009.

[2] Ştefan Andrei and Cristian Masalagiu. About the collatz conjecture. *Acta Informatica*, 35(2):167–179, 1998.

[3] Arthur Charguéraud. Pretty-big-step semantics. In *European Symposium on Programming*, pages 41–60. Springer, 2013.

[4] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. Precondition inference from intermittent assertions and application to contracts on collections. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 150–168. Springer, 2011.

[5] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, 1982.

[6] Markus Degen, Peter Thiemann, and Stefan Wehr. True lies: Lazy contracts for lazy languages (faithfulness is better than laziness). In *In 4. Arbeitstagung Programmiersprachen (ATPS09)*. Citeseer, 2009.

[7] Markus Degen, Peter Thiemann, and Stefan Wehr. The interaction of contracts and laziness. *Higher-Order and Symbolic Computation*, 25(1):85–125, 2012.

[8] Ewen Denney. Refinement types for specification. In *Programming Concepts and Methods PROCOMET98*, pages 148–166. Springer, 1998.

[9] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chetan Murthy, Catherin Parent, Christine Paulin-Mohring, and Benjamin Werner. *The COQ Proof Assistant: User's Guide: Version 5.6*. INRIA, 1992.

[10] Cormac Flanagan. Hybrid type checking. In *ACM Sigplan Notices*, volume 41, pages 245–256. ACM, 2006.

[11] Bastiaan J Heeren. *Top quality type error messages.* Utrecht University, 2005.

[12] David Himmelstrup. Interactive debugging with ghci. In *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 107–107. ACM, 2006.

[13] Ralf Hinze, Johan Jeuring, and Andres Löh. Typed contracts for functional programming. In *International Symposium on Functional and Logic Programming*, pages 208–225. Springer, 2006.

[14] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. *Deriving structural hylomorphisms from recursive definitions*, volume 31. ACM, 1996.

[15] Johan Jeuring, Alex Gerdes, and Bastiaan Heeren. Ask-elle: A haskell tutor. In *European Conference on Technology Enhanced Learning*, pages 453–458. Springer, 2012.

[16] BPY Lauwers. Contract inference for the ask-elle programming tutor. 2014.

[17] Barbara H Liskov and Jeannette M Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.

[18] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Conference on Functional Programming Languages and Computer Architecture*, pages 124–144. Springer, 1991.

[19] Bertrand Meyer. Eiffel: A language and environment for software engineering. *Journal of Systems and Software*, 8(3):199–246, 1988.

[20] Bertrand Meyer. *Design by contract*. Prentice Hall, 2002.

[21] Ulf Norell. Dependently typed programming in agda. In *Advanced Functional Programming*, pages 230–266. Springer, 2009.

[22] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[23] J Alan Robinson. Computational logic: The unification computation. *Machine intelligence*, 6(63-72):10–1, 1971.

[24] Patrick M Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *ACM SIGPLAN Notices*, volume 43, pages 159–169. ACM, 2008.

[25] Jurriën Stutterheim. Contract inferencing for functional programs. 2013.

[26] Niki Vazou, Eric L Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for haskell. In *ACM SIGPLAN Notices*, volume 49, pages 269–282. ACM, 2014.