Summary

Program errors are hard to detect and are costly, to both programmers who spend significant efforts in debugging, and for systems that are guarded by runtime checks. Static verification techniques have been applied to imperative and object-oriented languages, like Java and C#, for checking basic safety properties such as memory leaks. In a pure functional language, many of these basic properties are guaranteed by design, which suggests the opportunity for verifying more sophisticated program properties. Nevertheless, few automatic systems for doing so exist. In this thesis, we show the challenges and solutions to verifying advanced properties of a pure functional language, Haskell. We describe a sound and automatic static verification framework for Haskell, that is based on contracts and symbolic execution. Our approach gives precise *blame assignments* at compile-time in the presence of higher-order functions and laziness.

First, we give a formal definition of contract satisfaction which can be viewed as a denotational semantics for contracts. We then construct two contract checking wrappers, which are dual to each other, for checking the contract satisfaction. We prove the soundness and completeness of the construction of the contract checking wrappers with respect to the definition of the contract satisfaction. This part of my research shows that the two wrappers are projections with respect to a partial ordering *crashes-more-often* and furthermore, they form a projection pair and a closure pair. These properties give contract checking a strong theoretical foundation.

As the goal is to detect bugs during compile time, we symbolically execute the code constructed by the contract checking wrappers and prove the soundness of this approach. We also develop a technique named counter-example-guided (CEG) unrolling which only unroll function calls *on demand*. This technique speeds up the checking process.

Finally, our verification approach makes error tracing much easier compared with the existing set-based analysis. Thus equipped, we are able to tell programmers during compile-time *which function to blame and why* if there is a bug in their program. This is a breakthrough for lazy languages because it is known to be difficult to report such informative messages either at compile-time or run-time.

Acknowledgements

During the course of this research I benefited enormously from the knowledge and expertise of my advisers, Simon Peyton Jones and Alan Mycroft. I especially thank Simon Peyton Jones for his patient guidance.

I would like to thank following people for feedback and discussion: Koen Claesson, John Hughes, Matthias Blume, Matthias Felleisen, Robby Findler, Ralf Hinze, Rustan Leino, Byran Cook, Kenneth Knowles, Cormac Flanagan, Martin Berger, Kohei Honda, Matthew Parkinson, Viktor Vafeiadis, Colin Runciman as well as the anonymous referees for the Haskell Workshop 2006 and the ACM Symposium on Principles of Programming Languages (POPL) 2009.

This Ph.D. project is partly supported by Microsoft Research through its Ph.D. Scholarship Programme. I am grateful to have had opportunities to attend conferences and workshops during my Ph.D. studies. Besides Microsoft Research Cambridge, I would like to thank Lise Gough and Fabien Petitcolas for making those fruitful trips possible.

I would like to thank Daan Leijen and Wolfram Schulte for an internship at Microsoft Research Redmond. This provided an opportunity to widen my research in functional programming.

I would like to thank my friends from both Computer Laboratory and Churchill College for their friendship.

Most importantly, I would like to thank my parents for their love.

Contents

List of Figures

List of Definitions, Theorems and Lemmas

Part I

Chapter 1

Introduction

Program errors are common in software systems, including those that are constructed from functional languages. For greater software reliability, such errors should be reported accurately and detected early during program development. *Contract checking* (both static and dynamic) has been widely used in procedural and object-oriented languages [LN98, FLL+02, BCC+03, BLS04]. The difficulty of contract checking in functional languages lies in the presence of advanced features such as higher-order functions and laziness. However, dynamic checking of contracts for higher-order functions has been studied by [FF02, BM06, FB06, HJL06]. Recently, hybrid[1] contract checking [Fla06, KTG+06, KF07, GF07] for functional languages has also been proposed.

Inspired by the idea of the contract semantics [FF02, BM06], in this thesis, we describe a sound and automatic static verification tool for Haskell, that is based on contracts and symbolic execution. Our approach gives precise blame assignments at compile-time in the presence of higher-order functions and laziness. Consider:

```
f :: [Int] -> Int
f xs = head xs `max` 0
```

where `head` is defined in the module Prelude as follows:

```
head :: [a] -> a
head (x:xs) = x
head [] = error "empty list"
```

If we have a call `f []` in our program, its execution will result in the following error message from the runtime system of the Glasgow Haskell Compiler (GHC):

```
Exception: Prelude.head: empty list
```

This gives no information on which part of the program is wrong except that `head` has been wrongly called with an empty list. This lack of information is compounded by the

[1] A static contract checking followed by a dynamic contract checking.

fact that it is hard to trace function calling sequence at run-time for lazy languages, such as Haskell.

The programmers' intention is that `head` should not be called with an empty list. To achieve this, programmers can give a contract to the function `head`. Contracts are implemented as pragmas:

```
{-# CONTRACT head :: {s | not (null s)} -> {z | True} #-}
```

where `not` and `null` are just ordinary boolean-valued Haskell functions:

```
null :: [a] -> Bool
null [] = True
null xs = False

not True = False
not False = True
```

This contract places the onus on callers of `head` to ensure that the argument to `head` satisfies the expected precondition. With this contract, our compiler would generate the following error message (by giving a counter-example `(f [])`) when checking the definition of `f`:

```
Error: f []  calls head
             which fails head's precondition!
```

Suppose we change `f`'s definition to the following:

```
f xs = if null xs then 0
       else head xs `max` 0
```

With this correction, our compiler will not give any more warning as the precondition of `head` is now fulfilled.

Our goal is to detect crashes in a program where a *crash* is informally defined as an unexpected termination of a program (i.e. a call to `error`). Divergence (i.e. non-termination) is not considered to be a crash.

1.1 Contributions

In this thesis, we develop a compile-time checker to highlight a variety of program errors, such as pattern matching failure and integer-related violations (e.g. division by zero, array bound checks), that are common in Haskell programs. We make the following contributions:

1. Our system is the first static checker for a lazy functional language, intended for ordinary programmers. It has the following features:

- We check contract violation **statically** (like ESC/Java [FLL+02]), rather than dynamically (like run-time checking approaches [FF02, HJL06]).
- We check each program in a **modular** fashion on a per-function basis. We check the contract of a function f using mostly the contracts of functions that f calls, rather then by looking at their actual definitions. This modularity property is essential for the system to scale.
- **Contracts** are written **in Haskell** itself so that programmers do not need to learn a new language.
- We design a verification system for a **lazy** language. The framework can be easily tuned to verify programs written in a strict language, but not vice versa.
- We can detect and locate bugs in the presence of **higher-order functions** and **arbitrary data structures** (Chapter 4)

Few of these features are individually unique, but no system known to us offers them in combination.

2. We give a crisp, declarative specification for what it means for a term to satisfy a contract (Section 4.3), independent of the techniques used (theorem provers, run-time checks, whatever) to verify that it does indeed satisfy it. This is unusual, with the notable exception of Blume & McAllester's work [BM06]. However, unlike Blume & McAllester (and most other related work on higher-order contracts), we focus on *static* verification and target a *lazy* language. To the best of our knowledge, this is the first attempt for static checking of higher-order functions with contracts.

3. Our contracts themselves contain unrestricted Haskell terms. That means arbitrary functions can be used in contracts, including:

   - higher order functions
   - recursive functions
   - partial functions

   which are not supported by most automatic verification tools including popular ones such as ESC/Java [FLL+02] and Spec# [BLS04]. This hugely increases the expressiveness of the specification language and allows sophisticated properties to be conveniently expressed (Chapter 2). This also means we tackle head-on the question of what happens if the contract itself diverges (Section 5.2) or crashes (Section 5.3).

4. Despite these complications, we are able to give a very strong theorem expressing the soundness and completeness of contract wrappers as compared to contract satisfaction (Chapter 5). Our framework neatly accommodates some subtle points that others have encountered, including: ensuring that all contracts are inhabited (Section 4.3.4) and the `Any` contract (Section 4.3.3).

5. We develop a concise notation ($\triangleright$ and $\triangleleft$) for describing contract checking, which enjoys many useful properties (Section 5.5) for presenting a relatively-simple proof of contract wrappers (Chapter 6).

6. Unlike the traditional verification condition (VC) generation[2] (in some meta language) approach that solely relies on a theorem prover to verify it, we treat preconditions and postconditions as boolean-valued functions and check safety properties using symbolic simplification that adheres to Haskell's semantics instead (Section 7.1). This way, we have better control of the whole verification process and whenever necessary, we can ask an external theorem prover for assistance (Section 7.1.2).

7. We exploit a counter-example guided (CEG) unrolling technique to assist the symbolic simplification process (Section 7.2). CEG approach is used widely for abstraction refinement in the model checking community. However, to the best of our knowledge, this is the first time CEG is used in determining which function call to be unrolled.

8. We give a trace of functional calls that leads to a crash at compile-time, whilst such traces are usually offered by debugging tools at run-time. A counter-example is generated and reported together with its function calling trace as a warning message for each potential bug (Section 7.3).

9. We show that this symbolic-simplification approach is indeed sound w.r.t. the specification (unlike, say, ESC/Java) (Section 7.1). An induction approach used for contract checking of recursive functions is sound and we give formal definition and proof of soundness (Section 5.4). Our approach can verify advanced properties such as sorting (Section 8.3) and AVL trees (Section 8.5).

10. We integrate it to one branch of the Glasgow Haskell Compiler (GHC) so that the verification tool can deal with full Haskell. We evaluate our implementation on small but interesting real-life programs (Chapter 9).

1.2 Thesis Roadmap

This thesis is divided into three parts:

- **Appetiser**. Chapter 1 and 2 give a programmer's-eye view of the system and Section 2.3 describes the intuition of how static contract checking works. Section 1.3 (Technical Background) shows where we position ourselves in the area of program verification. Readers, who are familiar with program verification or bug detection, may skip Section 1.3.

- **Main Course**. Chapter 3–9 contains all the details of the static contract checking framework where Chapter 4-7 are the most substantial chapters of this thesis.

- **Dessert**. This thesis opens a new and fertile research area on static contract checking. Chapter 10 shows possible future work that can be done to enhance the current system. As there are many approaches in program verification, Chapter 11 justifies that our approach is new and plausible by comparing with closely related work in

[2]The computation of a VC is similar to the computation of a weakest precondition.

detail. We also suggest possible future collaboration with some of the related work. We put the related work chapter in Part III due to the fact that readers may have to understand many technical details of our system to appreciate the differences which may be small but crucial.

For the ease of reading, a page number reference to definitions, theorems and lemmas is given as a superscript.

1.3 Technical Background

Software reliability is a serious problem for modern society. We are in contact with software everyday, but this software often contains serious bugs. This is partly because software is getting more and more complex, and partly because program verification techniques are not advanced enough.

There are two reasons that make a functional language *the language of choice* for building complex and reliable software:

† It makes programming easier by introducing high-level features such as higher-order functions, abstract data types, polymorphism, laziness, etc.

‡ It avoids the side effects caused by pointers and aliasing which make imperative languages (e.g. C-like languages) much more error prone.

Nevertheless, functional programmers still spend tremendous time in debugging their programs. From the 2005 Glasgow Haskell Compiler (GHC) survey[3], the most-requested feature, after performance improvement, is some kind of debugger. About two decades after the design of ML/SML [MTH89] and one decade after the appearance of Haskell [Tea98], there is still no compiler that supports static automatic verification of these high-level languages. The trouble is that these high-level languages support advanced features (such as higher-order functions, complex recursions, laziness) that make programming easier, but make verification harder.

Program verification dates back to Hoare logic [Hoa69] and its extensions [EMC77, Apt81, DJ84, GCH89, Goe85]. Researchers are actively involved in automatically verifying imperative programs at compile-time. Some examples include ESC/Java [FS01, FLL+02] for Java, Spec♯ [BLS04] for C♯, and SLAM [BR02] for C. Formal reasoning for mutable data structures has also been studied, for example, Separation Logic [Rey02, ORY01] has been used in reasoning for low-level C-like languages. But most safety problems they try to tackle are avoided by the design of a functional language as mentioned earlier at the beginning of Section 1.3, hence very few results could be applied to verifying functional languages. We aim to study sophisticated safety properties of functional programs. In this thesis, we convert state monads to a core language which is similar to System F so that we do not have to handle states explicitly.

In this section, we give a brief overview of the world of correctness checking of functional programs along two axes:

[3] http://www.haskell.org/ghc/survey2005-summary.html

(1) level of rigorous in static checking (Section 1.3.1);

(2) compile-time to run-time (Section 1.3.2).

We only aim to give a general idea on the position of our work. More detailed comparisons of our work with other closely related works can be found in Chapter 11 (Related Work).

1.3.1 Degree of Static Verification

Static checking can improve software productivity because the cost of correcting an error is reduced dramatically if it is detected early. Figure 1.1 (adapted from [FLL+02]) compares static checkers on two important dimensions: the degree of error coverage obtained by running the tool and the cost of running the tool.

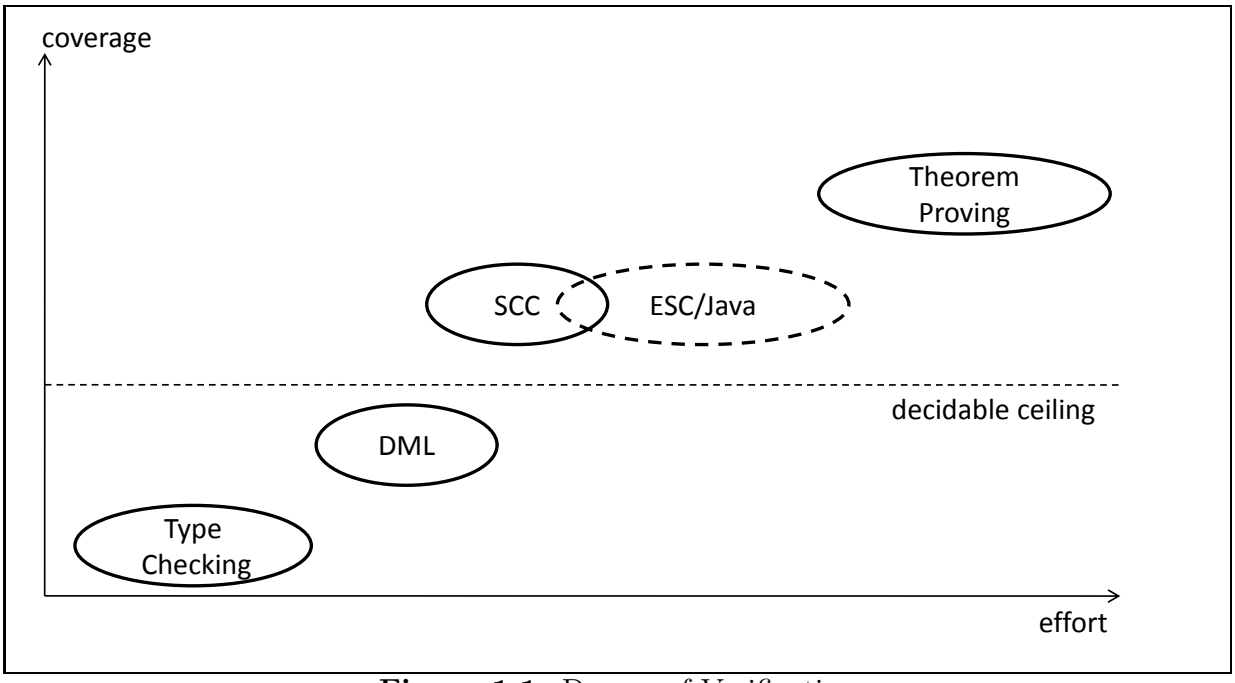

Figure 1.1: Degree of Verification

Our static contract checking (SCC) framework is close to the extended static checking (ESC) framework. However, existing ESC tools are all unsound (represented with dotted circle) while ours is sound.

At the lower left corner are the static checking techniques that are widely used, which require only modest effort, but catch only a limited class of errors, for example, conventional type checkers.

Another well-known static checking technique is dependent type checking, which is undecidable in general. However, if we restrict the constraints used in the dependent types to linear inequalities over integer domain, the dependent type checking is decidable. For example, a language that supports decidable dependent type checking is dependent ML (DML) [XP99].

At the top right corner are the most sophisticated program verification techniques which may cover all possible safety checking, for example, Isabelle/HOL [tea06b] and Coq [tea06a]. However, it may take both the theorem prover and programmers (who may have to supply necessary theorems) a great amount of effort to do the proof.

Hindley-Milner Type Checking

A type of a function is a general specification to the function. For example:

```
(+) :: Int -> Int -> Int
(/) :: Int -> Int -> Int
```

both functions `(+)` and `(/)` take two integers as input and return an integer as output. The type of the function does not specify what the function does (whether addition or division). A type checker reports an error during compile-time when it encounters an expression such as `1 + True` because the second argument `True` has type `Bool` which violates the required type `Int`. However, an expression `(5 + 0)` is safe while `(5 / 0)` will crash, though both of them are well-typed.

Dependent Type Checking over Restricted Constraint Domains

Dependent type checkers allow more constraints to be specified than conventional type checkers, for example:

```
append [] ys     = ys
append (x:xs) ys = x : append xs ys
 withtype {m,n:Nat} => [a](m) -> [a](n) -> [a](m+n)
```

The Hindley-Milner type of `append` is `[a] -> [a] -> [a]` which says that the function `append` takes two lists of elements of type `a` and return a list of type `a`. The dependent type `[a](m) -> [a](n) -> [a](m+n)` makes the original Hindley-Milner type *depend* on the value of `m` and `n` which refer to the length of each input list. The extra notation `{m,n:Nat}` says that the `m` and `n` are universally quantified and they denote natural numbers. So the dependent type of the function `append` says that the function takes two lists of length `m` and `n` respectively and return a list whose length is the sum of the lengths of the two input lists.

Consider the following functions:

```
(++) = append

rev [] = []
rev (x:xs) = rev xs ++ [x]
  withtype {m:Nat} => [a](m) -> [a](m)

length [] = 0
length (x:xs) = 1 + length xs
  withtype {m:Nat} => [a](m) -> Int(m)
```

The function `rev` reverses a list, the function `length` calculates the length of a list and the function `++` (which can be used in infix form) appends two lists. With the dependent types, assuming `xs` and `ys` are safe, a dependent type checker should be able to tell that the following expression

```
case length (rev (xs ++ ys)) == length (rev (ys ++ xs)) of
    True -> xs
    False -> error "huh"
```

is safe because `length (rev (xs ++ ys)) == length (rev (ys ++ xs))` always returns `True` and the call to "error" cannot be reached.

Extended Static Checking

Extended static checkers *extend* static type checkers by allowing more expressive constraints to be specified, so that they can catch more errors. The Extended Static Checking (ESC) approach shares the same goal as dependent type checking: to check more properties of a program than the basic type checking. Compare with DML, ESC relaxes the form of constraints to be verified. It allows arbitrary pure functions to be used in the specifications. Consider an example in ESC/Haskell, which reflects the general ESC style of annotation.

```
foo x y @ requires { prime x > sqrt y }
foo x y @ ensures  { $res == x*2     }
foo x y = case prime x > sqrt y of
           True -> x*2
           False -> error "foo"
```

where `$res` denotes the result of the function `foo`. We can see that arbitrary pure functions can be used in the specification so ESC is undecidable. In the above example, no tool can statically prove `prime x > sqrt y` for arbitrary `x` and `y`.

Tools that fall into this category include ESC/Modula-3 [LN98], ESC/Java [FLL+02], Spec# [BLS04] and ESC/Haskell [Xu06]. ESC/Modula-3 and ESC/Java are unsound while Spec# is sound because it requires invariants to be given. ESC/Haskell is sound and forms part of this thesis.

Theorem Proving

In the upper right corner of Figure 1.1 is full functional program verification, which theoretically catches all errors, but is extremely expensive. For example:

```
taut xs ys = case (rev (xs ++ ys) == rev ys ++ rev xs) of
              True -> xs
              False -> error "taut"
```

We know that if given two safe finite lists `xs` and `ys`, the test

```
rev (xs ++ ys) == rev ys ++ rev xs  -- A Theorem for finite lists!
```

should always evaluate to `True` because it is a tautology. However, in order to verify this tautology, a theorem prover may require programmers to provide some non-trivial lemmas. In the above case, a lemma stating the associativity of the function `++` is needed:

```
lemma assocAppend xs ys zs =
            (xs ++ ys) ++ zs == xs ++ (ys ++ zs)
```

The theorem prover has to prove the lemma based on the definition of `(++)` before applying the lemma to verify the theorem. Often, to prove one lemma, more lemmas have to be provided and proved. This whole process of proving one theorem can be very expensive.

Nevertheless, a theorem prover can be used as an assisting tool for static contract checking. This is illustrated in Section 7.1.2 where we use an external theorem prover to simplify expressions involving arithmetic.

1.3.2 Static Contract Checking vs Dynamic Contract Checking

Programmers can specify a property that they expect a function to have in the form of a contract. If all functions in a program satisfy their corresponding contracts, a program should not give any unexpected error during run-time. However, in general, not all contract violations can be detected during compile-time. An alternative approach is to check contract satisfaction at run-time and report failures if any run-time data, that a function takes, violates the function's contract. This approach is called *dynamic contract checking*. Findler and Felleisen [FF02] adopt this approach and have given a *dynamic* contract checking algorithm for Scheme, an untyped strict functional language. Research on dynamic assertion checking includes [VOS+05, HJL06, CL07]. However, dynamic checking suffers from two drawbacks. First, it consumes cycles that could otherwise perform useful computation. More seriously, dynamic checking provides only limited coverage - specifications are only checked on data values and code paths of actual executions. Thus, dynamic checking often results in incomplete and late detection of defects.

Flanagan [Fla06] has proposed a *hybrid contract checking* scheme: a static contract checking followed by a dynamic contract checking. (He uses the name *hybrid type* in [Fla06] because it refers to hybrid refinement type.) Hybrid contract checking can detect defects statically (whenever possible) and dynamically (only when necessary).

In this thesis, we focus on static contract checking and we can turn contracts that cannot be checked statically into dynamic contract checks. However, this is easy in the strict setting and is non-trivial in the lazy setting. Some work on lazy assertions [CMR03, CH06] has been proposed, but there are still some difficult open problems left to be solved. We will elaborate more in Section 10.6.

Chapter 2

Overview of Static Contract Checking

The type of a function constitutes a partial specification to the function. For example, `inc :: Int -> Int` says that `inc` is a function that takes an integer and returns an integer. A *contract* of a function gives more detailed specification. For example: `{-# CONTRACT inc :: {x | x > 0} -> {r | r > x} #-}` says that the function `inc` takes a positive value and returns a value that is greater than the input. A contract can therefore be viewed as a refinement to a type, so it is also known as refinement type in [FP91, Dav97, Fla06].

This thesis describes a system that allows a programmer to write a contract on some (but, like type signatures, not necessarily all) definitions, and then statically checks whether the definition *satisfies* the contract. This check is undecidable, and our system may give the result "definitely satisfies", "definitely does not satisfy", or "don't know". In the latter two cases we emit information that helps to localise the (possible) bug. We begin, however, by giving the flavour of contracts themselves with various examples. Section 4 gives formal semantics of contracts.

2.1 Expressiveness of the Specification Language

Consider a simple example:

```
div :: Float -> Float -> Float
div x y = case y == 0 of
           True  -> error "divide by zero"
           False -> x / y
```

where the operator (`/`) does the primary division job. The function `div` crashes when taking an argument that is equal to 0. Programmers can give the function `div` a contract:

```
{-# CONTRACT div :: {x | True} -> {y | y /= 0} -> {z | True} #-}
```

The contract of `div` says that the first argument can be any number (indicated by the weakest constraint `True`) and the second argument should not be zero; if these requirements are satisfied, the function should produce a number (again, we do not care what number it is). With this contract declaration, the compiler can tell (`div 5 0`) is a bug without exploring the definition of `div`.

Recall the earlier example:

```
{-# CONTRACT inc :: {x | x > 0} -> {z | z > x} #-}
```

We see that the `x` in the precondition is used in the postcondition. Here, we assume that the scope of `x` includes the RHS of `->` so that we can relate the input and the output of a function.

We now show the expressiveness of contracts with examples; each subsection focuses on a particular feature.

2.1.1 Recursive Functions Called in Contracts

Programmers often find that they use a data type with many constructors, but at some specialised contexts in the program only a subset of these constructors is expected to occur. Such a data type can also be recursive. For example, in a software module of the Glasgow Haskell Compiler (GHC) that is used after type checking, we may expect that types would not contain mutable type variables. Under such a scenario, certain constructor patterns may be safely ignored. We use a simple example to illustrate such scenario by defining a datatype `T` and a predicate `noT1` as follows.

```
data T = T1 Bool | T2 Int | T3 T T

noT1 :: T -> Bool
noT1 (T1 _) = False
noT1 (T2 _) = True
noT1 (T3 t1 t2) = noT1 t1 && noT1 t2
```

The function `noT1` returns `True` when given any data structure of type `T` in which there is no data node with a `T1` constructor. We may have a consumer:

```
sumT :: T -> Int
{-# CONTRACT sumT :: {x | noT1 x} -> {z | True} #-}
sumT (T2 a) = a
sumT (T3 t1 t2) = sumT t1 + sumT t2
```

which requires that the input data structure does not contain any `T1` node. We may also have a producer like:

```
rmT1 :: T -> T
{-# CONTRACT rmT1 :: {x | True} -> {z | noT1 z} #-}
rmT1 (T1 a) = case a of
                True -> T2 1
                False -> T2 0
rmT1 (T2 a) = T2 a
rmT1 (T3 t1 t2) = T3 (rmT1 t1) (rmT1 t2)
```

We know that for all crash-free `t` of type `T`, a call `(sumT (rmT1 t))` will not crash. Thus, by allowing a recursive predicate (e.g. `noT1`) to be used in the contracts, we can achieve such a goal.

2.1.2 Higher-Order Functions Called in Contracts

Now consider a higher-order function `filter` whose result is asserted with the help of another recursive higher-order function `all`.

```
filter :: (a -> Bool) -> [a] -> [a]
{-# CONTRACT filter :: {f | True} -> {x | True} -> {z | all f z} #-}
filter f [] = []
filter f (x:xs') = case (f x) of
                     True -> x : filter f xs'
                     False -> filter f xs'

all :: (a -> Bool) -> [a] -> Bool
all f [] = True
all f (x:xs) = f x &&  all f xs

(&&) True x = x
(&&) False x = False
```

Note that in the contract of `filter`, the variable `f` in the parameter contract can be used in the result contract. In general, we assume the scope of bound variables in contracts extends over the RHS of the `->`.

2.1.3 Contracts for Higher Order Function Parameters

The contract notation is more expressive than the `requires`, `ensures` notation used in our initial work [Xu06], because it scales properly to higher order functions. Consider an example adapted from [BM06]:

```
f1 :: (Int -> Int) -> Int
{-# CONTRACT f1 :: ({x | True} -> {y | y >= 0}) -> {z | z >= 0} #-}
f1 g = (g 1) - 1

f2 = f1 (\x -> x - 1)
```

The contract of `f1` says that if `f1` takes a function, which returns a natural number when given any integer, the function `f1` itself returns a natural number.

The Findler-Felleisen algorithm in [FF02] (a dynamic contract checking algorithm) can detect a violation of the contract of `f1`, however, it cannot tell the argument of `f1` in the definition of `f2` fails `f1`'s precondition due to lack of evidence during run-time. On the other hand, the Sage system in [KTG+06] (a hybrid contract checking system) can detect the failure in `f2` statically, and can report contract violation of `f1` at run-time. Our system can report both failures at compile-time with the following informative messages:

```
Error: f1's postcondition fails
       because (g 1) >= 0 does not imply
               (g 1) - 1 >= 0

Error: f2 calls f1
       which fails f1's precondition
```

2.1.4 Functions without Contracts

A special feature of our system is that it is not necessary for programmers to annotate all the functions. There are two reasons why a programmer may choose not to annotate a function with contracts:

1. The programmer is lazy.
2. There is no contract that is more compact than the function definition itself.

Examples of the second case are the function `(&&)`, `null` and even a recursive function like `noT1` in Section 2.1.

If a function, say f, which may be recursive, does not have a contract annotation, we assume programmers want to check whether f satisfies the trivial contract `{x | True}`.

It is possible to infer simple contracts for non-recursive functions, such as `head`, by collecting conditions that do not leading to a crash. It is much harder to infer contracts for recursive functions. Contract inference is not in the scope of this thesis; we discuss some existing work on specification inference in Section 11.5.

2.1.5 Laziness

A conservative contract may cause false alarms especially in the presence of laziness. For example:

```
fst (a,b) = a
f3 xs = (null xs, head xs)
f4 xs = fst (f3 xs)
```

We could give `f3` the following contract:

```
{-# CONTRACT f3 :: {xs | not (null xs)} -> {z | True}  #-}
```

With this contract, our system may report the following error message when checking the definition of `f4`.

```
Error: (f4 []) fails f3's precondition
```

However, the call `fst (f3 xs)` is safe in a lazy language even if `xs` has value `[]` because the call to `head []` will not be invoked.

One way to reduce such false alarms is to inline `f3` and `fst` so that we have `fst (f3 xs)` simplified to `null xs` and we know `f4` is safe. Although inlining can reduce false alarms due to laziness, if the size of the lazy function is big, or the function is recursive, the inlining strategy breaks down. For example:

```
fstN :: (Int, Int) -> Int -> Int
fstN (a, b) n = case n > 0 of
                  True  -> fstN (a + 1, b) (n - 1)
                  False -> a

g2  = fstN (5, error "fstN") 100
```

We need to inline `fstN` for 100 times to know `g2` is safe.

A better way to reduce the false alarms due to laziness is to introduce a special contract `Any`, which every expression satisfies. We can give function `fstN` the following contract:

```
{-# CONTRACT fstN :: ({x | True}, Any) -> {n | True} -> {z | True} #-}
```

The contract of `fstN` says that it does not care what the second component of the argument is, as long as the first component is crash-free, the result is crash-free. Here, with the contract `Any`, without inlining any function, our system can tell that `g2` is safe.

This means we give a crash `(error "msg")` a contract `Any`, while in [BM06] an expression that unconditionally crashes satisfies no contract. This is one of the key differences in designing the contract semantics.

2.1.6 Data Constructor Contract

In Section 2.1.5, we gave `fst`'s argument the contract `({x | True}, Any)`; that is, the argument should be a pair whose first component satisfies `{x | True}`, and whose second satisfies `Any`. We generalise this form for any user-defined data constructor so that programmers can give a contract to the sub-components of any data constructor. For example, we can use the list constructor (:) to create a contract like this:

```
{x | x > 0} : {xs | all (<0) xs}
```

which says that the first element in the list is positive while the rest are all negative. We may also have this:

```
{x | x > 0} : {y | y > 0} : Any
```

which says that the first two elements are positive while the rest can be anything (i.e. may crash). There are two things to note:

1. The contract `Any` and the contract `{x | True}` are different: all expressions satisfy `Any` while only crash-free expressions satisfy `{x | True}`. The difference is explained in detail in Section 4.3 and Section 5.1.2.

2. Although we can give a contract to a component of a data structure, it is different from a recursive contract (Section 10.2).

We allow any user-defined data constructors to be used in declaring a contract. For example:

```
data A = A1 Int Bool | A2 A

f5 :: A -> Int
{-# CONTRACT f5 :: A1 {x | x > 0} {y | y == True} -> {z | z > x} #-}
f5 (A1 x y) = case y of
                True  -> x + 1
                False -> error "f4"
```

As we allow data constructors to be used in contracts, we can replace the contract `{y | y == True}` by `True` as `True` itself is a nullary constructor. There are two things to note:

1. In the contract of `f5`, the data constructor `A1` is used, whereas in the type specification the data type `A` is used.

2. A call `(f5 (A2 ...))` fails the precondition of `f5`.

Moreover, data constructor `A2` can be used in constructing contracts as well. For example:

```
{-# CONTRACT f6 :: A2 {x | f5 x > 0} -> {z | True} #-}
```

Function `f6` expects an input satisfying `(A2 (A1 {x | x > 0} True))`. Note that the constructor contract only specifies properties for a top-level data constructor. To specify properties recursively over a data structure, we need a *recursive contract*, which is one of our future enhancements (Section 10).

2.1.7 Partial Functions in Contracts

A partial function is one that may *crash* or *diverge*. For example, function `head`, which crashes when given an argument `[]`. Since we allow arbitrary Haskell code in contracts, what are we to say about contracts that crash or diverge? One possibility is to simply exclude all such contracts – but excluding divergence (in a statically-checkable system) requires a termination checker, and excluding functions like `head` is extremely restrictive. For example:

```
head :: [a] -> a
{-# CONTRACT head :: {x | not (null x)} -> {z | True} #-}
head (x:xs) = x
head []     = error "empty list"

headPlus :: [Int] -> Int
{-# CONTRACT headPlus :: {xs | not (null xs)}
                      -> {z | z > head xs} #-}
headPlus []    = error "Urk"
headPlus (x:xs) = x+1
```

Here the postcondition uses `head` (which may crash), but that seems entirely reasonable in view of the precondition that `xs` is non-empty. Nevertheless, such a contract is rejected by [BM06], because of the call to `head`.

Our approach is to permit divergence in contracts (which avoids the requirement for a termination checker), but to require them to be "*crash-free*". Our definition of crash-free-ness for *contracts* takes account of dependency, and hence is much more liberal than requiring each Haskell *term* in the contract to be independently crash-free (which excludes `head`). This liberality is, we believe, key to making contracts usable in practice. We discuss crash-freeness of contracts in §5.3.1 and divergence in §5.2.1.

2.1.8 Contract Synonym

In previous sections, we used the contract `{x | True}` at many places. In our system, we allow programmers to define contract synonyms which are similar to the idea of type synonyms. For example, we may have:

```
{-# CONTRACT Ok = {x | True} #-}
{-# CONTRACT Pos = {x | x > 0} #-}
{-# CONTRACT Nat = {x | x >= 0} #-}
{-# CONTRACT NotNull = {xs | not (null xs)} #-}

{-# CONTRACT head :: NotNull -> Ok #-}
head (x:xs) = x
```

In this thesis, a contract synonym is just a shorthand. In future, we may allow contract synonyms to have parameters.

2.2 Three Outcomes from Our System

Some properties that our system may attempt to check can either be undecidable or difficult to verify at compile-time. For example:

```
g1 :: Int -> Int
{-# CONTRACT  g1 :: Ok -> Ok #-}
g1 x = case (prime x > square x) of
       True -> x
       False -> error "g1"
```

where `prime` gives the `x`th prime number and `square` gives $\texttt{x}^2$. Most theorem provers including ours are unable to tell the condition `prime x > square x` always holds or not (in fact, it does not hold), so we report a potential crash. For another example:

```
g2 :: [a] -> [a] -> [a]
{-# CONTRACT g2 :: Ok -> Ok -> Ok #-}
g2 xs ys = case (rev (xs ++ ys) == rev ys ++ rev xs) of
            True -> xs
            False -> error "g2"
```

Some theorem provers may be able to prove the validity of the theorem:

```
rev (xs ++ ys) == rev ys ++ rev xs
```

for all well-defined `xs` and `ys`. However, this is often at high cost and may require extra lemmas from programmers such as the associativity of the append operator `++`.

As it is known to be expensive to catch all errors in a program, our system chooses only to provide meaningful messages to programmers based on three possible outcomes after checking for potential crashes for each function definition (say f). They are:

(a) **Definitely safe**. If the precondition of f is satisfied, any call to f with crash-free arguments will not crash.

(b) **Definite bug**. Any call to f with crash-free arguments, satisfying the declared precondition of f, crashes or loops.

(c) **Possible bug**. The system cannot decide which of (a) or (b) is the case.

For the last two cases, a trace of function calls that leads to a (potential) crash together with a counter-example[1] will be generated and reported to the programmer. We make a distinction between definite and possible bugs, in order to show the urgency of the former and also because the latter may not be a real bug.

[1]Programmers can set the number of counter-examples they would like to see.

2.3 The Plan for Verification

It is all very well for programmers to *claim* that a function satisfies a contract, but how can we *verify* that claim statically (i.e. at compile time)? The usual approach is to extract verification conditions (VC) from the program that faithfully embody the semantics of the language and send those VCs to a theorem prover. If we get answer"Yes", we know a function satisfies its contract. But if we get answer "No", it is hard to tell which function to blame and why.

Our overall plan, which is similar to that of Blume and McAllester [BM06], is as follows.

- Our overall goal is to prove that the program does not *crash*, so we must first say what programs are, and what it means to "crash" (Chapter 3).
- Next, we give a semantic specification for what it means for a function f to "satisfy a contract" t, written $f \in t$ (Chapter 4).
- From a function definition $f = e$ we form a term $e \triangleright t$ pronounced "e ensures t". This term behaves just like e except that

  (a) if e disobeys t then the term crashes;

  (b) if the context uses e in a way not permitted by t then the term loops.

  The term $e \triangleright t$ is essentially the wrapper mechanism first described by Findler and Felleisen [FF02], with some important refinements (Chapter 5).
- With these pieces in place, we can write down our main theorem (Chapter 5), namely that

  $$e \in t \quad \Longleftrightarrow \quad (e \triangleright t) \text{ is crash-free}$$

  We must ensure that everything works properly, even if e diverges, or laziness is involved, or the contract contains divergent or crashing terms.
- Using this theorem, we may check whether $f \in t$ holds as follows: we attempt to prove that $(e \triangleright t)$ is crash-free — that is, does not crash under all contexts. We conduct this proof in a particularly straightforward way: we perform symbolic evaluation of $(e \triangleright t)$. If we can simplify the term to a new term e', where e' is syntactically safe — that is, contains no crashes everywhere in the expression — then we are done. This test is sufficient, but not necessary; of course, the general problem is undecidable.

Part II

Chapter 3

The Language

The language presented in this thesis, named language $\mathcal{H}$, is simply-typed lambda calculus with case-expression, constructors and integers. Language $\mathcal{H}$ is simpler than the language we use in our implementation, which is the GHC Core Language [Tea98], which is similar to System F and includes parametric polymorphism.

3.1 Syntax

The syntax of our language $\mathcal{H}$ is shown in Figure 3.1. A program is a module that contains a set of data type declarations and function definitions. Expressions include variables, type and term abstractions, type and term applications, constructors and `case` expressions. We treat `let`-expressions as syntactic sugar:

$$\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \quad \equiv_s \quad (\lambda x.e_2)\ e_1$$

We omit local `letrec` as well, we only have recursive (or mutually recursive) top-level functions. We introduce a special function $\texttt{fin}_n$, which is only for internal usage (Section 5.2.1). Readers can ignore the $\texttt{fin}_n$ for the moment. There are two *exception values* adopted from [Xu06]:

`BAD` is an expression that *crashes*. A program crashes if and only if it evaluates to `BAD`. For example, a user-defined function `error` can be explicitly defined as:

```
error :: String -> a
error s = BAD
```

A preprocessor ensures that source programs with missing cases of pattern matching are explicitly replaced by the corresponding equations with `BAD` constructs. For example, after preprocessing, function `head`'s definition becomes:

```
head (x:xs) = x
head []     = BAD
```

pgm	$\in$	**Program**	
pgm	$:=$	$def_1, \ldots, def_n$	
def	$\in$	**Definition**	
def	$:=$	$decl$	
	$\mid$	$f \in t$	contract attribution
	$\mid$	$f\ \vec{x} = e$	top-level definition
$decl$	$\in$	**Data Type**	
$decl$	$:=$	$\texttt{data}\ T\ \vec{\alpha}\ \texttt{where}$	data type decl
		$\overrightarrow{K \in \vec{\tau_i} \to T\ \vec{\alpha}}$	data constructors
	$\mid$	$\texttt{data}\ T\ \vec{\alpha}\ =$	data type decl
		$K_1\ \vec{\tau_i} \mid \cdots \mid K_n\ \vec{\tau_i}$	data constructors
x, y, v, f, g	$\in$	**Variables**	
a, e, p	$\in$	**Exp**	**Expression**
a, e, p	$::=$	$v \mid \lambda(x :: \tau).e \mid e_1\ e_2$	
	$\mid$	$\texttt{case}\ e_0\ \texttt{of}\ (v :: \tau)\ \ alts$	case-expression
	$\mid$	$K\ \vec{e}$	constructor
	$\mid$	$\texttt{fin}_n\ e$	finite evaluation
	$\mid$	r	exception
r	$::=$	$\texttt{BAD}$	a crash
	$\mid$	$\texttt{UNR}$	unreachable
$alts$	$::=$	$alt_1 \ldots alt_n$	
alt	$::=$	$pt\ \to\ e$	case alternative
pt	$::=$	$K\ (x_1 :: \tau_1) \ldots (x_n :: \tau_n)$	pattern
	$\mid$	$\texttt{DEFAULT}$	
τ	$\in$	**Types**	
τ	$\mid$	$\texttt{Int} \mid \texttt{Bool} \mid () \mid \ldots$	base types
	$\mid$	T	data type
	$\mid$	α	type variable
	$\mid$	$\forall \alpha.\tau$	
val	$\in$	**Value**	
val	$::=$	$n \mid K\ \vec{e} \mid \lambda x \in t.e \mid \texttt{UNR} \mid \texttt{BAD}$	

Figure 3.1: Syntax of the Language $\mathcal{H}$

`UNR` (short for "unreachable") is an expression that gets stuck. This is *not* considered as a crash, although the execution comes to a halt without delivering a result. A

program that loops forever also does not crash, and does not deliver a result, so you can think of `UNR` as a term that simply goes into an infinite loop.

The exceptional values are for internal usage and hidden from Haskell programmers. Their behaviour is made precise by the operational semantics in Figure 3.3.

The top-level declaration $f \in t$ is the claim that f satisfies contract t. We discuss contracts in Chapter 4.

3.2 Type Checking Rules for Expression

The language $\mathcal{H}$ is statically typed in the conventional way. Figure 3.1 gives the syntax of types, while Figure 3.2 gives type checking rules. A type judgement has the form

$$\Delta \vdash e :: \tau$$

which states that given Δ (which is a mapping from variable to its type, contract and definition), e has type τ assuming that any free variable in it has type given by Δ. If $\Delta = \emptyset$, we omit the Δ, and write $\vdash e :: \tau$.

$$\Delta \vdash \texttt{BAD} :: \tau \quad \text{[T-BAD]} \qquad \Delta \vdash \texttt{UNR} :: \tau \quad \text{[T-UNR]}$$

$$\frac{v :: \tau \in \Delta}{\Delta \vdash v :: \tau} \quad \text{[T-Var]} \qquad \frac{K :: \vec{\tau} \to T \in \Delta \qquad \Delta \vdash \vec{e} :: \vec{\tau}}{\Delta \vdash K\ \vec{e} :: T\ \vec{\alpha}} \quad \text{[T-Con]}$$

$$\frac{\Delta \vdash e :: \texttt{Bool}}{\Delta \vdash \texttt{fin}_n\ e :: \texttt{Bool}} \quad \text{[T-fin]} \qquad \frac{\Delta, x :: \tau_1 \vdash e :: \tau_2}{\Delta \vdash (\lambda(x :: \tau_1).e) :: \tau_1 \to \tau_2} \quad \text{[T-Lam]}$$

$$\frac{\Delta \vdash e_1 :: \tau_1 \to \tau_2 \qquad \Delta \vdash e_2 :: \tau_1}{\Delta \vdash (e_1\ e_2) :: \tau_2} \quad \text{[T-App]}$$

$$\frac{\Delta \vdash e_0 :: T\ \vec{\tau} \qquad \Delta, \{v :: T\ \vec{\tau}\}, \overrightarrow{\{K_i\ \vec{x_i} :: T\ \vec{\tau}\}} \vdash e_i :: \tau}{\Delta \vdash (\texttt{case}\ e_0\ \texttt{of}\ (v :: T\ \vec{\tau})\ \{K_i\ \vec{x_i} \to e_i\}) :: \tau} \quad \text{[T-Case]}$$

$$\frac{\Delta \vdash \lambda x.e_0 :: \tau_1 \to \tau_2 \qquad \Delta \vdash e :: \tau}{\Delta \vdash (\texttt{case}\ \lambda x.e_0\ \texttt{of}\ (v :: \tau_1 \to \tau_2)\ \{\texttt{DEFAULT} \to e\}) :: \tau} \quad \text{[T-CaseLam]}$$

Figure 3.2: Type Checking Rules

As we do type checking before contract checking, we assume all expressions are well-typed (i.e. no type error) in the rest of this thesis. Note that nothing substantial in the thesis depends delicately on the type system. The reason we ask that programs are well-typed is to avoid the technical inconvenience in designing the semantics of contracts if, say, evaluation finds an ill-typed expression `(3 True)`.

3.3 Operational Semantics

The semantics of the language $\mathcal{H}$ is given by the confluent, non-deterministic rewrite rules in Figure 3.3. The language is confluent because it is a subset of the untyped lambda calculus which is confluent. We use a small-step reduction-rule semantics, rather than (say) a deterministic more machine-oriented semantics, because the more concrete the semantics becomes, the more involved the proofs become too.

Most of these rules are entirely conventional. The rule [E-top] deals with a top-level function call f. We fetch its definition from the environment Δ, which maps a variable to its type, contract and definition. To save clutter, we usually leave this environment implicit, rather than writing (say) $\Delta \vdash e_1 \rightarrow_M e_2$.

Evaluation proceeds by repeatedly replacing the current redex with its corresponding one-step reduction until a value is reached. (Note that `BAD` and `UNR` are considered values.) Rule [E-ctx] allows a reduction step to take place anywhere. The expression $\mathcal{C}[\![e]\!]$ means substituting the $\bullet$ in the context $\mathcal{C}$ by the expression e (i.e. $\mathcal{C}[e/\bullet]$). The relation $e_1 \rightarrow e_2$ performs a single step reduction and the relation $\rightarrow^*$ is the reflexive-transitive closure of $\rightarrow$.

The unconventional features are the "M" subscript on the reduction arrow, the form $\texttt{fin}_n\ e$, and the reduction rules [E-fin1,2,3]. Their job is to convert a boolean-valued divergent expression to `True` before the fuel M is used up. These aspects all concern contracts containing divergent expressions, and are discussed in detail in Section 5.2.1, where we define $\rightarrow^*$ in terms of $\rightarrow^*_M$. For the moment, we can simply ignore the subscripts and `fin`.

The rule [E-beta] performs the standard β-reduction. When a scrutinee of a `case` expression is a data constructor that matches one of the patterns, it is also a redex, shown in the rule [E-match1]. If the scrutinee does not match any pattern pt_i except the `DEFAULT` branch, then the `DEFAULT` branch will be taken as shown in the rule [E-match2], In the rule [E-match3], if the scrutinee is a function and the only branch is `DEFAULT`, the RHS of the branch is taken. The rule is only useful when we introduce a function `` `seq` `` in Section 5.1.1. In the rule [E-match4], if the scrutinee does not match any pattern and there is no `DEFAULT` branch, indicated by $(K\ \vec{a} \not\sim pt_i)$, we return `UNR`. This relates to the fact that during preprocessing we fill in all missing branches by `BAD`, and now we would like to use `UNR` to indicate a missing branch. The purpose of doing so is to make the symbolic execution less cluttered. We discuss symbolic execution in detail in Section 7.1. Rules [E-exapp] and [E-excase] deal with exception values in the usual way.

Now we can give the usual definition of contextual equivalence:

Definition 1 (Semantically Equivalent) *Two expressions e_1 and e_2 are semantically equivalent, namely $e_1 \equiv_s e_2$, iff*

$$\forall \mathcal{C}.\quad \mathcal{C}[\![e_1]\!] \rightarrow^* \texttt{BAD} \quad \Longleftrightarrow \quad \mathcal{C}[\![e_2]\!] \rightarrow^* \texttt{BAD}$$

Two expressions are said to be semantically equivalent, if under all closing contexts, if one evaluates to `BAD`, the other also evaluates to `BAD`. The conventional definition on

Evaluation

$$\frac{(f = \lambda x.e) \in \Delta}{f \to_M \lambda x.e} \text{ [E-top]} \qquad \frac{e \to_M e' \quad n < M}{\texttt{fin}_n\ e \to_M \texttt{fin}_{n+1}\ e'} \text{ [E-fin1]}$$

$$\begin{array}{rcll} \texttt{fin}_n\ \texttt{UNR} & \to_M & \texttt{True} & \text{[E-fin2]} \\ \texttt{fin}_n\ val & \to_M & val & \text{[E-fin3]} \\ & & (val \neq \texttt{UNR} \text{ and } n < M) & \\ \texttt{fin}_M\ e & \to_M & \texttt{True} & \text{[E-fin4]} \end{array}$$

$$(\lambda x.e_1)\ e_2 \to_M e_1[e_2/x] \quad \text{[E-beta]}$$

$$\texttt{case}\ K_i\ \vec{y_i}\ \texttt{of}\ \{\ldots;\ K_i\ \vec{x_i} \to e_i;\ \ldots\} \to_M e_i[y_i/x_i] \quad \text{[E-match1]}$$

$$\texttt{case}\ K\ \vec{a}\ \texttt{of}\ \{pt_i \to e_i;\ \texttt{DEFAULT} \to e\} \to_M e \text{ for all } i.\ K\ \vec{a} \not\sim pt_i \quad \text{[E-match2]}$$

$$\texttt{case}\ \lambda x.e_0\ \texttt{of}\ \{\texttt{DEFAULT} \to e\} \to_M e \quad \text{[E-match3]}$$

$$\texttt{case}\ K\ \vec{a}\ \texttt{of}\ \{pt_i \to e_i\} \to_M \texttt{UNR} \text{ for all } i.\ K\ \vec{a} \not\sim pt_i \quad \text{[E-match4]}$$

$$\begin{array}{rcll} r\ e & \to_M & r & \text{[E-exapp]} \\ \texttt{case}\ r\ \texttt{of}\ alts & \to_M & r & \text{[E-excase]} \end{array}$$

$$\frac{e_1 \to_M e_2}{\mathcal{C}[\![e_1]\!] \to_M \mathcal{C}[\![e_2]\!]} \text{ [E-ctx]}$$

Contexts

$$\begin{array}{rcl} \mathcal{C} & ::= & \bullet \mid \mathcal{C}\ e \mid e\ \mathcal{C} \mid \lambda x.\mathcal{C} \mid K\ e_1 \ldots \mathcal{C}_i \ldots e_n \\ & \mid & \texttt{case}\ \mathcal{C}\ \texttt{of}\ alts \\ & \mid & \texttt{case}\ e\ \texttt{of}\ \{p_1 \to e_1; \ldots \\ & & \quad ; p_i \to \mathcal{C}_i; \ldots \\ & & \quad ; p_n \to e_n\} \end{array}$$

Strict Context

$$\mathcal{S} ::= \bullet \mid S\ e \mid \texttt{case}\ S\ \texttt{of}\ alts$$

Figure 3.3: Semantics of the Language $\mathcal{H}$

semantical equivalence uses `()` (unit) or any value that is syntactically comparable, for

example, `True`, `False`, etc. That is:

$$\forall \mathcal{C}.\quad \mathcal{C}[\![e_1]\!] \to^* () \quad \Longleftrightarrow \quad \mathcal{C}[\![e_2]\!] \to^* ()$$

However, if we use this definition together with our operational semantics, we cannot distinguish the two exceptional values `BAD` and `UNR`. So instead of using `()`, we choose another syntactically comparable value `BAD`.

Lemma 1 (Equivalence) *For all (possibly open) expressions e_1, e_2, if $e_1 \to^* e_2$, then $e_1 \equiv_s e_2$.*

As mentioned earlier, there exists an implicit environment that maps variables to its type, contract and definition. So Lemma 1 actually says "if $\Delta \vdash e_1 \to^* e_2$, then $\Delta \vdash e_1 \equiv_s e_2$".

Lemma 2[p44], which is only used in the proof of Lemma 5[p46], says that the strictness of a context does not change the behaviour of an expression. This implies that if we can prove a theorem that holds for strict context, then the theorem holds for all contexts.

Lemma 2 (Strict Context)

$$\forall \mathcal{S},\ \texttt{BAD} \notin_s \mathcal{S},\ \mathcal{S}[\![e]\!] \not\to^* \texttt{BAD} \iff \forall \mathcal{C},\ \texttt{BAD} \notin_s \mathcal{C},\ \mathcal{C}[\![e]\!] \not\to^* \texttt{BAD}$$

Proof We prove two directions separately.
($\Rightarrow$) We prove it by induction on the size of context. We only have to examine those non-strict context one by one:

1. Case $C = e'\ C'$: Since $\texttt{BAD} \notin_s \mathcal{C}$, $\texttt{BAD} \notin_s e'$. That means $e' \not\to^* \texttt{BAD}$. By inspecting rules in Figure 3.3, `BAD` can only be caught by the rule [E-exapp]. By induction hypothesis, $\forall \mathcal{C}', \texttt{BAD} \notin_s \mathcal{C}', \mathcal{C}'[\![e]\!] \not\to^* \texttt{BAD}$. Since $e' \not\to^* \texttt{BAD}$, $\mathcal{C}'[\![e]\!] \not\to^* \texttt{BAD}$, $\texttt{BAD} \notin_s e'$ and $\texttt{BAD} \notin_s \mathcal{C}'$, we have $\mathcal{C}[\![e]\!] \not\to^* \texttt{BAD}$ as desired.

2. Case $C = \lambda x.C'$: It is a lambda value, so $\mathcal{C}[\![e]\!] \not\to^* \texttt{BAD}$.

3. Case $C = K\ e_1 \ldots C'_i \ldots e_n$: It is a constructor value, so $\mathcal{C}[\![e]\!] \not\to^* \texttt{BAD}$.

4. Case $C = \texttt{case}\ e'\ \texttt{of}\ \{p_1 \to e_1; \ldots p_i \to C'_i; \ldots; p_n \to e_n\}$: Since $\texttt{BAD} \notin_s \mathcal{C}$, $\texttt{BAD} \notin_s e'$. That means $e' \not\to^* \texttt{BAD}$. By inspecting rules in Figure 3.3, `BAD` can only be caught by rule [E-excase]. By induction hypothesis, $\forall \mathcal{C}', \texttt{BAD} \notin_s \mathcal{C}', \mathcal{C}'[\![e]\!] \not\to^* \texttt{BAD}$. Since $e' \not\to^* \texttt{BAD}$, $\mathcal{C}'[\![e]\!] \not\to^* \texttt{BAD}$, $\texttt{BAD} \notin_s e'$ and $\texttt{BAD} \notin_s \mathcal{C}'$, we have $\mathcal{C}[\![e]\!] \not\to^* \texttt{BAD}$ as desired.

5. Case $C = \lambda x.C'$: It is a lambda value, so $\mathcal{C}[\![e]\!] \not\to^* \texttt{BAD}$.

($\Leftarrow$) Immediate because a strict context S is a subcontext of C. ∎

3.4 Crashing

We use `BAD` to signal that something has gone wrong in the program: it has *crashed*. A program is correct if and only if the `main` function in a program does not *crash*.

Definition 2 (Crash) *A closed expression* e crashes *iff* $e \rightarrow^* \texttt{BAD}$.

Our technique can only guarantee *partial* correctness. A diverging program does not crash.

Definition 3 (Diverges) *A closed expression* e diverges*, written* $e\uparrow$*, iff either* $e \rightarrow^* \texttt{UNR}$*, or there is no value val such that* $e \rightarrow^* val$.

At compile-time, one easy way to check the safety of a program is to see whether the program is syntactically safe:

Definition 4 (Syntactic safety) *A (possibly-open) expression* e *is* syntactically safe *iff* $\texttt{BAD} \notin_s e$. *Similarly, a context* $\mathcal{C}$ *is syntactically safe iff* $\texttt{BAD} \notin_s \mathcal{C}$.

The notation $\texttt{BAD} \notin_s e$ means `BAD` does not syntactically appear anywhere in e, similarly for $\texttt{BAD} \notin_s \mathcal{C}$. For example, $\lambda x.x$ is syntactically safe while $\lambda x.\ (\texttt{BAD}, x)$ is not. An expression with free variables is not considered as syntactically safe.

Definition 5 (Crash-free Expression) *A (possibly-open) expression* e *is crash-free iff*

$$\forall \mathcal{C}.\ \texttt{BAD} \notin_s \mathcal{C},\ \vdash \mathcal{C}[\![e]\!] :: (),\ \mathcal{C}[\![e]\!] \not\rightarrow^* \texttt{BAD}$$

The notation $\vdash \mathcal{C}[\![e]\!] :: ()$ means $\mathcal{C}[\![e]\!]$ is closed and well-typed under the type system shown in Figure 3.2. The Definition 5[p45] says that if an expression does not crash in all safe contexts, which are like probes for `BAD`, then the expression cannot crash regardless whether there is any `BAD` syntactically appearing in it because all of them are unreachable. That means a crash-free expression may not be syntactically safe, for example:

```
\x -> case x * x >= 0 of
        True -> x + 1
        False -> BAD
```

The tautology $x * x >= 0$ is always true, so the `BAD` can never be reached. For another example, $(\texttt{BAD}, 3)$ is not crash-free because there exists a context $(\texttt{fst}\ \bullet)$, such that:

$$\texttt{fst}\ (\texttt{BAD}, 3) \rightarrow \texttt{BAD}$$

In short, crash-freeness is a *semantic* concept, and hence undecidable, while syntactic-safety is *syntactic* and readily decidable. Certainly, a syntactically safe expression is crash-free and crash-freeness is preserved during execution.

Lemma 3 (Syntactically Safe Expression is Crash-free) *For all* e,

$$e \text{ is syntactically safe} \quad \Rightarrow \quad e \text{ is crash-free}$$

PROOF Given $\texttt{BAD} \notin_s e$, for all $\mathcal{C}$ such that $\texttt{BAD} \notin_s \mathcal{C}$, we know $\texttt{BAD} \notin_s \mathcal{C}[\![e]\!]$. Recall the operational semantics in Figure 3.3, in order to introduce a BAD at RHS of $\rightarrow$, we must have a BAD at the LHS of $\rightarrow$. Since $\texttt{BAD} \notin_s \mathcal{C}[\![e]\!]$, we have $\mathcal{C}[\![e]\!] \not\rightarrow^* \texttt{BAD}$. ∎

Lemma 4 (Crash-free Preservation) *Given* $e_1 \rightarrow e_2$,

$$e_1 \textit{ is crash-free} \iff e_2 \textit{ is crash-free}$$

PROOF We prove two directions by contradiction.
($\Rightarrow$)
Suppose e_2 is not crash-free. By Definition 5[p45] (Crash-free Expression), there exists a $\mathcal{C}$ such that $\texttt{BAD} \notin_s \mathcal{C}$ and $\mathcal{C}[\![e_2]\!] \rightarrow^* \texttt{BAD}$. By [E-ctx] and $e_1 \rightarrow e_2$ and $\mathcal{C}[\![e_2]\!] \rightarrow^* \texttt{BAD}$, we have: $\mathcal{C}[\![e_1]\!] \rightarrow^* \mathcal{C}[\![e_2]\!] \rightarrow^* \texttt{BAD}$. As we know e_1 is crash-free, we reach contradiction. Thus, we are done.
($\Leftarrow$)
Suppose e_1 is not crash-free. By Definition 5[p45] (Crash-free Expression), there exists a $\mathcal{C}$ such that $\texttt{BAD} \notin_s \mathcal{C}$ and $\mathcal{C}[\![e_1]\!] \rightarrow^* \texttt{BAD}$. By [E-ctx] and $e_1 \rightarrow e_2$ and confluence of the language, we have $\mathcal{C}[\![e_2]\!] \rightarrow^* \texttt{BAD}$. With the assumption that e_2 is crash-free, we reach contradiction. Thus, we are done. ∎

The forward direction of Lemma 5[p46] cannot be derived directly from the definition of crash-free expression (Definition 5[p45]), which requires the context to be syntactically safe. In the proof of Lemma 5[p46] ($(\Rightarrow)$ direction), we use an operator $\lfloor . \rfloor$ which replaces all BADs in an expression by UNR. We call it *neutering*, which is recursively defined in Figure 3.4. The neutering operator satisfies the Lemma 6[p47].

$$\begin{array}{rcl}
\lfloor \lfloor e \rfloor \rfloor & = & \lfloor e \rfloor \\
\lfloor \texttt{BAD} \rfloor & = & \texttt{UNR} \\
\lfloor \texttt{UNR} \rfloor & = & \texttt{UNR} \\
\lfloor e_1\ e_2 \rfloor & = & \lfloor e_1 \rfloor\ \lfloor e_2 \rfloor \\
\lfloor \lambda v.e \rfloor & = & \lambda v.\lfloor e \rfloor \\
\lfloor K\ e_1 \ldots e_n \rfloor & = & K\ \lfloor e_1 \rfloor \ldots \lfloor e_n \rfloor \\
\lfloor \texttt{case}\ e_0\ \texttt{of}\ \{pt_i \rightarrow e_i\} \rfloor & = & \texttt{case}\ \lfloor e_0 \rfloor\ \texttt{of}\ \{pt_i \rightarrow \lfloor e_i \rfloor\}
\end{array}$$

Figure 3.4: Neutering Expression and Contract

Lemma 5 (Crash-free Function) *For all (possibly-open) terms* $\lambda x.e$,

$$\lambda x.e \textit{ is crash-free}$$
$$\iff$$
$$\textit{for all (possibly-open) crash-free } e',\ e[e'/x] \textit{ is crash-free.}$$

PROOF We prove two directions separately.
($\Rightarrow$)

$$\begin{array}{ll}
 & \lambda x.e \text{ is crash-free} \\
\Rightarrow & (\text{By Lemma } 6^{\text{p47}},\ e' \text{ is crash-free } \Rightarrow \lfloor e' \rfloor \equiv_s e' \\
 & \text{and by the definition of crash-free expression}) \\
 & \text{for all crash-free } e',\ e[e'/x] \text{ is crash-free}
\end{array}$$

($\Leftarrow$) We have the following proof.

$$
\begin{array}{ll}
 & \forall e', e[e'/x] \text{ is crash-free} \\
\Rightarrow & \text{(By Lemma } 4^{\text{p46}}\text{)} \\
 & \forall e', (\lambda x.e)\ e' \text{ is crash-free} \\
\Rightarrow & \text{(By Definition } 5^{\text{p45}} \text{ (Crash-free Expression))} \\
(1) & \forall e', \forall \mathcal{C},\ \texttt{BAD} \notin_s \mathcal{C},\ \mathcal{C}[\![(\lambda x.e)\ e']\!] \not\rightarrow^* \texttt{BAD} \\
\Rightarrow & \text{(By Lemma } 2^{\text{p44}} \text{ (Strict Context))} \\
(2) & \forall e', \forall \mathcal{D}',\ \texttt{BAD} \notin_s \mathcal{D}',\ \mathcal{D}'[\![(\lambda x.e)\ e']\!] \not\rightarrow^* \texttt{BAD} \\
\Rightarrow & \text{(reasoning at (*) below)} \\
(3) & \forall \mathcal{D},\ \texttt{BAD} \notin_s \mathcal{D},\ \mathcal{D}[\![(\lambda x.e)]\!] \not\rightarrow^* \texttt{BAD} \\
\Rightarrow & \text{(By Lemma } 2^{\text{p44}} \text{ (Strict Context))} \\
 & \forall \mathcal{C},\ \texttt{BAD} \notin_s \mathcal{C},\ \mathcal{C}[\![(\lambda x.e)]\!] \not\rightarrow^* \texttt{BAD} \\
\Longleftrightarrow & \text{(By Definition } 5^{\text{p45}} \text{ (Crash-free Expression))} \\
 & \lambda x.e \text{ is crash-free}
\end{array}
$$

(*) To prove (3), we appeal to Lemma 2^{p44}, which allows us to examine only strict contexts. There are 3 cases to consider:

- Case $\mathcal{C} = \bullet$. Since $(\lambda x.e)$ is a value, $(\lambda x.e) \not\rightarrow^* \texttt{BAD}$.
- Case $\mathcal{C} = \bullet\ e''$. By (2) where we choose e' as e'', we are done.
- Case $\mathcal{C} = \texttt{case}\ \bullet\ \texttt{of}\ alts$. Since $\lambda x.e$ is not a constructor, $\texttt{case}\ \lambda x.e\ \texttt{of}\ alts$ cannot be further reduced, so $(\texttt{case}\ \lambda x.e\ \texttt{of}\ alts) \not\rightarrow^* \texttt{BAD}$.

End of proof. ∎

Lemma 6 (Neutering) *If e is crash-free, then $\lfloor e \rfloor \equiv_s e$.*

Proof Since e is crash-free, all BADs in e are not reachable so by converting all BADs in e to UNR by $\lfloor . \rfloor$ does not change the semantics of e. Formally, we prove this by induction on reduction rules. ∎

3.5 Behaves-the-same

We now define an ordering, named *Behaves-the-same*, which is useful in later sections.

Definition 6 (Behaves the same) *Expression e_1 behaves the same as e_2 w.r.t. a set of exceptions R, written $e_1 \ll_R e_2$, iff for all contexts $\mathcal{C}$, such that $\forall i \in \{1, 2\}.\ \vdash \mathcal{C}[\![e_i]\!] :: ()$*

$$\mathcal{C}[\![e_2]\!] \rightarrow^* r \in R \quad \Rightarrow \quad \mathcal{C}[\![e_1]\!] \rightarrow^* r$$

Definition 6^{p47} says that e_1 either behaves the same as e_2 or throws an exception from R. (The definition does not look as strong as that, but as every theorist knows, it is. For example, could e_1 produce True while e_2 produces False? No, because we could find

a context $\mathcal{C}$ that would make $\mathcal{C}[\![e_2]\!]$ throw an exception while $\mathcal{C}[\![e_1]\!]$ does not.) In our framework, there are only two exceptional values in R: `BAD` and `UNR`. Certainly, if e_2 itself throws an exception, then e_1 must throw the same exception.

As we only have two exceptional values `BAD`, `UNR` (which are dual to each other) in R, this yields Lemma 7[p48]. We omit $\{\}$ if there is only one element in R.

Lemma 7 (Properties of Behaves-the-same) *For all closed e_1 and e_2,*

$$e_1 \ll_{\texttt{UNR}} e_2 \iff e_2 \ll_{\texttt{BAD}} e_1$$

Proof We prove two directions separately.

($\Rightarrow$) We have the following proof:

$$\begin{array}{ll} & e_1 \ll_{\texttt{UNR}} e_2 \\ \iff & \text{(By defn of } \ll_{\texttt{UNR}}) \\ & \forall \mathcal{C}.\ \mathcal{C}[\![e_2]\!] \to^* \texttt{UNR} \Rightarrow \mathcal{C}[\![e_1]\!] \to^* \texttt{UNR} \\ \iff & \text{(By logic)} \\ & \forall \mathcal{C}.\ \mathcal{C}[\![e_1]\!] \not\to^* \texttt{UNR} \Rightarrow \mathcal{C}[\![e_2]\!] \not\to^* \texttt{UNR} \end{array}$$

We want to show that $\forall \mathcal{D}.\ \mathcal{D}[\![e_1]\!] \to^* \texttt{BAD} \Rightarrow \mathcal{D}[\![e_2]\!] \to^* \texttt{BAD}$.

Assume $\mathcal{D}[\![e_1]\!] \to^* \texttt{BAD}$.

Let $\mathcal{C} = \texttt{case } (\texttt{fin } \mathcal{D}[\![\bullet]\!]) \texttt{ of } \{\texttt{DEFAULT} \to \texttt{UNR}\}$

Now we have $\mathcal{C}[\![e_1]\!] \to^* \texttt{BAD} \Rightarrow \mathcal{C}[\![e_2]\!] \not\to^* \texttt{UNR}$.

Since $\mathcal{C}[\![e_2]\!] = \texttt{case } \mathcal{D}[\![e_2]\!] \texttt{ of } \{\texttt{DEFAULT} \to \texttt{UNR}\}$, we have $\mathcal{D}[\![e_2]\!] \to^* \texttt{BAD}$.

So we have

$$\forall \mathcal{D}.\ \mathcal{D}[\![e_1]\!] \to^* \texttt{BAD} \Rightarrow \mathcal{D}[\![e_2]\!] \to^* \texttt{BAD}$$

($\Leftarrow$) By replacing `BAD` by `UNR` and `UNR` by `BAD` in the above proof for the direction ($\Rightarrow$), we get the proof for the direction ($\Leftarrow$). ∎

3.6 Crashes-more-often

We now study the specialized ordering *crashes-more-often*, which plays a crucial role in proving Property 1[p63].

Definition 7 (Crashes-more-often) *An expression e_1 crashes more often than e_2, written $e_1 \preceq e_2$, iff $e_1 \ll_{\texttt{BAD}} e_2$.*

Informally, e_1 crashes more often than e_2 if they behave in exactly the same way except that e_1 may crash when e_2 does not. By Definition 7[p48], Lemma 7[p48] also says that:

$$e_1 \ll_{\texttt{UNR}} e_2 \iff e_2 \preceq e_1$$

Theorem 1 (Crashes-more-often is AntiSymmetric) *For all expressions e_1 and e_2, $e_1 \preceq e_2$ and $e_2 \preceq e_1$ iff $e_1 \equiv_s e_2$.*

PROOF It follows immediately from the definition of $\equiv_s$ (Definition 1[p42]) and the definition of $\preceq$. ∎

The crashes-more-often operator has many properties. Lemma 8[p49] says that BAD crashes-more-often then all expressions; all expressions crash more often then a diverging expression. Lemma 9[p49] gives more intuitive properties.

Lemma 8 (Properties of Crashes-more-often - I)

$$\begin{array}{ll} (a) & \texttt{BAD} \preceq e_2 \\ (b) & e_1 \preceq e_2 \quad \textit{if } e_2\uparrow \end{array}$$

PROOF We prove each property separately (all by contradiction) and we assume type soundness.

(a) Assume there exists a context $\mathcal{C}$ such that $\mathcal{C}[\![e_2]\!] \rightarrow^* \texttt{BAD}$ and $\mathcal{C}[\![\texttt{BAD}]\!] \not\rightarrow^* \texttt{BAD}$. There are two possibilities for $\mathcal{C}[\![e_2]\!] \rightarrow^* \texttt{BAD}$: (1) the BAD is from the context $\mathcal{C}$; (2) the BAD is from the hole e_2. For case (1), we must have $\mathcal{C}[\![\texttt{BAD}]\!] \rightarrow^* \texttt{BAD}$ since we use the same context $\mathcal{C}$. For case (2), if the hole is evaluated, we reach BAD immediately. So we reach a contradiction and we are done.

(b) Given $e_2\uparrow$, assume there exists a context $\mathcal{C}$ such that $\mathcal{C}[\![e_2]\!] \rightarrow^* \texttt{BAD}$ and $\mathcal{C}[\![e_1]\!] \not\rightarrow^* \texttt{BAD}$. Since $e_2\uparrow$ and $\mathcal{C}[\![e_2]\!] \rightarrow^* \texttt{BAD}$, we know the BAD is from the context $\mathcal{C}$. So no matter what e_1 is, we have $\mathcal{C}[\![e_1]\!] \rightarrow^* \texttt{BAD}$. Thus, we again reach a contradiction and we are done. ∎

Lemma 9 (Properties of Crashes-more-often - II) *If $e_1 \preceq e_2$*

$$\begin{array}{lrcl} (a) & e_1 \rightarrow^* K\ \overline{f_1} & \Rightarrow & e_2 \rightarrow^* K\ \overline{f_2} \textit{ or } e_2\uparrow \\ (b) & e_1\uparrow & \Rightarrow & e_2\uparrow \\ (c) & e_1 \textit{ is crash-free} & \Rightarrow & e_2 \textit{ is crash-free} \\ (d) & e_1 \rightarrow^* \lambda x.e_1' & \Rightarrow & e_2 \rightarrow^* \lambda x.e_2' \textit{ or } e_2\uparrow \end{array}$$

PROOF We prove each property separately (all by contradiction):

(a) Given $e_1 \rightarrow^* K\ \overline{f_1}$, assume neither $e_2 \rightarrow^* K\ \overline{f_2}$ nor $e_2\uparrow$. Then we must have $e_2 \rightarrow^* \texttt{BAD}$. By the definition of $\preceq$ and the fact that $e_1 \preceq e_2$, if $e_2 \rightarrow^* \texttt{BAD}$, then $e_1 \rightarrow^* \texttt{BAD}$. Since $e_1 \rightarrow^* K\ \overline{f_1}$, we reach a contradiction and we are done.

(b) Given $e_1\uparrow$, assume $e_2 \not\downarrow$. Then $e_2 \rightarrow^* val$ and there exists a syntactically safe context $\mathcal{C}$ such that $\mathcal{C}[\![e_2]\!] \rightarrow^* \texttt{BAD}$. But $\mathcal{C}[\![e_1]\!]$ always diverges as e_1 diverges if $\texttt{BAD} \notin_s \mathcal{C}$. By the fact that $e_1 \preceq e_2$ and by the definition of $\preceq$, we reach a contradiction and we are done.

(c) Given e_1 is crash-free, assume e_2 is not crash-free. By Definition 5^{p45} (Crash-free Expression), there exists a syntactically safe context $\mathcal{C}$ such that $\mathcal{C}[\![e_2]\!] \rightarrow^* \texttt{BAD}$. By the fact that $e_1 \preceq e_2$ and by the definition of $\preceq$, we have $\mathcal{C}[\![e_1]\!] \rightarrow^* \texttt{BAD}$. This contradicts with another assumption that e_1 is crash-free. Since we reach a contradiction, we are done.

(d) The proof is similar to that in (a). ∎

Chapter 4

Contracts and Their Semantics

Having discussed the language of programs, we now discuss the language of contracts.

4.1 Syntax

t	$\in$	**Contract**	
t	$::=$	$\{x \mid p\}$	Predicate Contract
	$\mid$	$x\colon t_1 \to t_2$	Dependent Function Contract
	$\mid$	(t_1, t_2)	Tuple Contract
	$\mid$	`Any`	Polymorphic Any Contract

Figure 4.1: Syntax of Contracts

The syntax of contracts is given in Figure 4.1. A predicate contract $\{x \mid p\}$ can be viewed as a boolean-valued function $\lambda x.p$ where p is arbitrary expression in $\mathcal{H}$. We use the syntax $x\colon t_1 \to t_2$ for a dependent function contract where x can be used in t_2. If x is not used in t_2, it can be omitted. We adopt this notation from [Aug98, Fla06], which is equivalent to $\Pi x\colon t_1 \to t_2$. For example:

```
{-# CONTRACT inc :: x':{x | x > 0} -> {r | r > x'} #-}
inc x = x + 1
```

We abbreviate $x'\colon \{x \mid p\} \to t$ to $\{x \mid p\} \to t[x/x']$. So the contract of `inc` can be written as:

```
{-# CONTRACT inc :: {x | x > 0} -> {r | r > x} #-}
```

by assuming the scope of `x` includes the RHS of the `->`. In general, given $x\colon t_1 \to t_2$, the x can be omitted unless t_1 is a function type, for example:

```
{-# CONTRACT f3 :: k:({x | x > 0} -> {z | z > x})
                -> {y | True} -> {r | k y} #-}
```

Here, we cannot omit `k`, which denotes a function contract, if it is used in the RHS of `->`. Moreover, the variables x and z cannot be used outside the brackets.

In Chapter 2, we have seen an example of using tuple contract. For reasons of notational simplicity, we only restrict data constructor contracts to pairs only in this thesis, but the idea generalises readily. In our full implementation, we deal with arbitrary user-defined data constructor contracts.

We introduce a special polymorphic contract named `Any` which every expression satisfies. Section 4.3.3 shows why it is important to have `Any`.

4.2 Type Checking for Contracts

As the predicates in contracts are just boolean-valued Haskell expressions, it might be believed that we would simply call Haskell's type-checker to type-check the pre/postcondition. However, there are a few interesting issues to consider. For example:

```
head :: [a] -> a
head (x:xs) = x
```

Suppose a programmer gives this precondition for `head`:

```
{-# CONTRACT head :: {xs | xs /= []} -> Ok #-}
```

As the inequality operator `/=`, which is defined in type class `Eq`, is used, it makes the precondition stronger than necessary. That means it implicitly requires the type of `head` to be `head :: Eq a => [a] -> a`. This means during the type checking of contracts, we need to bear in mind that we should reject those contracts that cause the type of the function to be stronger.

$$\frac{\Delta, a :: k \vdash_c t :: \tau}{\Delta \vdash_c (\forall a :: k.\ t) :: \tau} \quad \text{[C-Forall]}$$

$$\frac{}{\Delta \vdash_c \texttt{Any} :: \tau} \quad \text{[C-Any]}$$

$$\frac{\Delta, x :: \tau \vdash_c e :: \texttt{Bool}}{\Delta \vdash_c \{x \mid e\} :: \tau} \quad \text{[C-One]}$$

$$\frac{\Delta \vdash_c c_1 :: \tau_1 \qquad \Delta, x :: \tau_1 \vdash_c c_2 :: \tau_2}{\Delta \vdash_c x : c_1 \to c_2 :: \tau_1 \to \tau_2} \quad \text{[C-Fun]}$$

$$\frac{\Delta \vdash_c c_i :: \tau_i \text{ for } i = 1, 2}{\Delta \vdash_c (c_1, c_2) :: (\tau_1, \tau_2)} \quad \text{[C-Tuple]}$$

Figure 4.2: Type Checking Rules for Contract

A contract type judgement has the form

$$\Delta \vdash_c t \in \tau$$

which states that given Δ (a mapping from program variable to its type, and from type variable to its kind), e has type τ assuming that any free variable in it has type given by Δ. Contract type checking rules are shown in Figure 4.2. In our full implementation, the Δ also contains the type classes (TC) in scope and we need an additional typing rule as follows.

$$\frac{\Delta, TC\ a \vdash_c t :: \tau}{\Delta \vdash_c t :: TC\ a \Rightarrow \tau} \quad \text{[C-TyClass]}$$

4.3 Contract Satisfaction

We give the semantics of contracts by defining "e satisfies t", written $e \in t$, in Figure 4.3. This is a purely declarative specification of contract satisfaction, it says *which* terms satisfy a contract, without saying *how* a satisfaction check might be performed. We regard the ability to give a simple, declarative, programmer-accessible specification of contract satisfaction as very important, but it is a property that few related works share, with the notable and inspiring exception of [BM06], as that paper says:

> The structure of a non-compositional semantics like [the Findler-Felleisen wrapping algorithm] is difficult to understand. With just Definition 1 [which says that a term satisfies a contract if its wrapping cannot crash] to hand, an answer to the question "Does e satisfy t?" is not easy because it involves consideration of every possible context. Nor can we ignore this problem, since in our experience most people's intuition differs from [Definition 1].

In Figure 4.3, both e and t may mention functions bound in the top-level definitions Δ. These functions are necessary for the evaluation relation of rule [A1] to make sense. To reduce clutter, we do not make these top-level bindings explicit, by writing $\Delta \vdash e \in t$, but instead allow rule [E-top] of Figure 3.3 to consult Δ implicitly.

Given $e :: \tau$ and $\vdash t :: \tau$, we define $e \in t$ as follows:

$$e \in \{x \mid p\} \iff e\uparrow \text{ or } (e \text{ is crash-free and } p[e/x] \not\rightarrow^* \{\texttt{BAD}, \texttt{False}\}) \quad \text{[A1]}$$

$$e \in x\colon t_1 \to t_2 \iff e\uparrow \text{ or } (e \to^* \lambda x.e' \text{ and } \forall e_1 \in t_1.\ (e\ e_1) \in t_2[e_1/x]) \quad \text{[A2]}$$

$$e \in (t_1, t_2) \iff e\uparrow \text{ or } (e \to^* (e_1, e_2) \text{ and } e_1 \in t_1 \text{ and } e_2 \in t_2) \quad \text{[A3]}$$

$$e \in \texttt{Any} \iff \texttt{True} \quad \text{[A4]}$$

Figure 4.3: Contract Satisfaction

4.3.1 Predicate Contract

In [A1] we say an expression e has contract $\{x \mid p\}$ written $e \in \{x \mid p\}$, we mean e either diverges or it is a crash-free expression that satisfies the predicate p in the contract. The predicate p may give four possible outcomes: $\uparrow$, `True`, `False` and `BAD` (four-valued logic). We consider$\uparrow$ and `True` to be safe while the outcomes `False` and `BAD` to be unsafe. To say e satisfies p, we require $p[e/x] \not\to^* \{\texttt{BAD}, \texttt{False}\}$, which means $p[e/x]\uparrow$ or $p[e/x] \to^* \texttt{True}$.

The alert reader will notice that [A1] specifies that *only crash-free terms satisfy a predicate contract* $\{x \mid p\}$. This means that the contract $\{x \mid \texttt{True}\}$, which we abbreviate to `Ok`, is satisfied precisely by the crash-free terms. For example:

$$\lambda x.x \;\in\; \texttt{Ok} \qquad (3,5) \;\in\; \texttt{Ok} \qquad \lambda x.(\texttt{BAD}, x) \;\notin\; \texttt{Ok}$$

Other choices are possible, but we postpone the discussion to Section 5.1.2, when we have more scaffolding in place.

Only diverging expressions satisfy contracts $\{x \mid \texttt{False}\}$ and $\{x \mid \texttt{BAD}\}$. For example:

$$\begin{array}{lcllcl} \texttt{UNR} & \in & \{x \mid \texttt{False}\} & \texttt{bot} & \in & \{x \mid \texttt{False}\} \\ \texttt{UNR} & \in & \{x \mid \texttt{BAD}\} & \texttt{bot} & \in & \{x \mid \texttt{BAD}\} \end{array}$$

where `bot` is defined as:

```
bot = bot
```

We elaborate more on divergence in Section 4.3.4.

4.3.2 Dependent Function Contract and Tuple Contract

In [A2], we say an expression e has dependent function type $x\colon t_1 \to t_2$, when e is applied to any argument that satisfies the contract t_1, it produces a result that satisfies the contract t_2. To get *dependent* function contracts we must simply remember to substitute $[e_1/x]$ in t_2.

In [A3], if an expression evaluates to a tuple, we expect each subcomponent satisfies its corresponding contract. A tuple expression, which is not crash-free, can satisfy only a tuple contract. For example:

$$\begin{array}{lcl} (\texttt{BAD}, 3) & \notin & \{x \mid (\texttt{snd}\ x) > 0\} \\ (\texttt{BAD}, 3) & \in & (\texttt{Any}, \{x \mid x > 0\}) \end{array}$$

However, we can have:

$$(\texttt{True}, 2) \in \{x \mid (\texttt{snd}\ x) > 0\}$$

As defined in [A1], a crash-free expression may satisfy a predicate contract.

4.3.3 Any Contract

If we only have [A1]-[A3], the expression `BAD` would not satisfy any contract. In a lazy language this is much too conservative, so in [A4] we introduce a special contract, named `Any`, which is satisfied by any expression including `BAD`. For example:

```
{-# CONTRACT fst :: (Ok, Any) -> Ok #-}
fst (x,y) = x
```

`Any` is also useful in post-conditions: a function whose postcondition is `Any` is a function that may crash. Haskell programmers often write packaged versions of Haskell's `error` function, such as

```
myError :: String -> a
{-# CONTRACT myError :: Ok -> Any #-}
myError s = error ("Fatal error: " ++ s)
```

So `BAD` satisfies `Any`. In fact, `BAD` satisfies *only* the contract `Any` because it fails the constraints stated in [A1]-[A3]:

$$\begin{array}{rcl} \texttt{BAD} & \notin & (\texttt{Any}, \texttt{Any}) \\ \texttt{BAD} & \notin & \texttt{Any} \to \texttt{Any} \end{array}$$

Lemma 10[p55] says that the only contract that `BAD` satisfies is `Any`.

Lemma 10 (Contract `Any`) *If* $\texttt{BAD} \in t$, *then* $t = \texttt{Any}$.

Proof By inspecting the definition of $\in$, the only contract that `BAD` satisfies is `Any`. ∎

4.3.4 Diverging Terms

The definitions in Figure 4.3 specify that a divergent term e satisfies *every* contract. This choice is expressed directly for predicate contracts $\{x \mid p\}$ and tuple contracts (t_1, t_2), and is an easy consequence for function contracts. We made this choice because otherwise we would often have to prove termination in order to prove that $e \in t$. For example:

```
f x = case x < 10 of
        True -> x
        False -> f (x/2)
```

Does `f` $\in$ `Ok` $\to \{x \mid x < 10\}$? The `True` branch clearly satisfies the postcondition but what about the `False` branch? Specifying that divergence satisfies any contract allows us to answer "yes" without proving termination. Furthermore, despite divergence, a caller of `f` can still rely on `f`'s postcondition:

```
g y = case (f y > 10) of
        True -> error "Urk"
        False -> True
```

Here `g` cannot crash, because `f` guarantees a result less than 10, or else diverges.

Our choice has the nice consequence that *every contract is inhabited* (by divergence). This matters. Consider whether ($\lambda x.$`BAD`) satisfies $\{x \mid$ `False`$\} \rightarrow$ `Ok`. If $\{x \mid$ `False`$\}$ was uninhabited, the answer would be "yes". But that choice is incompatible with building a rigorous connection (sketched in Section 2.3) between contract satisfaction and Findler-Felleisen-style wrapping. Indeed, Findler and Blume are forced to invent an awkward (and entirely informal) predicate form "`non-empty-predicate`" [FB06], which we do not need.

4.4 Contract Satisfaction for Open Expressions

We have mentioned that e and t may mention functions bound in the top-level environment. These functions participate in the evaluation of rule [A1]. But suppose that the programmer declares:

```
{-# CONTRACT f :: {x | x > 0} -> Ok #-}
f = ...
```

When checking the contract of a function `g` that calls `f`, we should presumably assume only `f`'s declared contract, *without looking at its actual definition.* Doing so is more modular, and allows the programmer to leave room for future changes by specifying a contract that is more restrictive than the current implementation.

This goal is easily achieved. Suppose the declared contracts for `f` and `g` are t_f, t_g respectively, and the definition of `g` is `g` $= e_g$ where `f` is called in e_g. Then, instead of checking that $e_g \in t_g$, we check that

$$(\lambda f.\ e_g) \ \in \ t_f \rightarrow t_g$$

That means we simply lambda-abstract over any variables free in e_g that have declared contracts. This approach also allows the programmer to omit a contract specification (just as type signatures are often omitted), in which case the contract checker can "look inside" the definition when proving the correctness of calls to that function. The exact details are a software engineering matter; our point here is that the underlying infrastructure allows a variety of choices.

The same technique simplifies the problem of checking satisfaction for recursive functions. If the programmer specifies the contract t_f for a definition `f` $= e$, then it suffices to check that

$$\lambda f.e \ \in \ t_f \rightarrow t_f$$

which is easier because $\lambda f.e$ does not call `f` recursively. There is nothing new here – it is just the standard technique of loop invariants in another guise – but it is packaged very conveniently.

In other words, imagine we have a contract judgement:

$$\Delta \vdash e \in t$$

which states that given Δ, which is a mapping from variable to its type, contract and definition.

Definition 8 (Contract judgement) *We write $\Delta \vdash e \in t$ to mean that e has contract t assuming that any free variable in e has contract given by Δ and any free variable in t has definition given by Δ. Suppose $\Delta = \{f_1 \mapsto (\tau_1, t_1, e_1), \ldots, f_n \mapsto (\tau_n, t_n, e_n)\}$, we define:*

$$\Delta \vdash e \in t \iff \lambda f_1. \ldots . f_n . e \in t_1 \to \cdots \to t_n \to t$$

This means, in theory (i.e. in the formalization of the verification), we only need to deal with closed expressions; in practice (i.e. in the implementation), we may refer to the environment Δ when necessary.

4.5 Subcontract Relation

Definition 9 (Subcontract) *For all closed contracts t_1 and t_2, t_1 is a subcontract of t_2, written $t_1 \leqq t_2$, iff*

$$\forall e.\ e \in t_1 \Rightarrow e \in t_2$$

For open contracts t, we assume implicitly that there is an environment Δ, which is a mapping from variable to its type, contract and definition (See Definition 8[p57] in Section 4.4).

Moreover, the subcontract relation can be illustrated in rule-form shown in Figure 4.4. Each rule in Figure 4.4 is a theorem. The relation $p \Rightarrow_e q$ in rule [C-Pred] is defined in Definition 10. Rule [C-Any] follows directly from the definition of $\leqq$. We now study the rules [C-Pred], [C-DepFun] and [C-Tup]. We assume the statement above the line is true, and prove the statement below the line is true. We leave the proof of other direction as a open problem.

$$\frac{p \Rightarrow_e q}{\{x \mid p\} \leqq \{x \mid q\}}\ [\text{C-Pred}] \qquad \frac{t_1 \leqq t_3 \qquad t_2 \leqq t_4}{(t_1, t_2) \leqq (t_3, t_4)}\ [\text{C-Tup}]$$

$$\frac{t_3 \leqq t_1 \qquad \forall e \in t_3.\ t_2[e/x] \leqq t_4[e/x]}{x\colon t_1 \to t_2 \leqq x\colon t_3 \to t_4}\ [\text{C-DepFun}] \qquad t \leqq \texttt{Any}\ [\text{C-Any}]$$

Figure 4.4: Subcontract Relation

Definition 10 (Boolean Expression Implication) *For all boolean expressions p and q, we say p implies q (written $p \Rightarrow_e q$) iff* $\begin{pmatrix} \texttt{case } p \texttt{ of} \\ \texttt{True} \to \texttt{BAD} \\ \texttt{False} \to () \end{pmatrix} \preceq \begin{pmatrix} \texttt{case } q \texttt{ of} \\ \texttt{True} \to \texttt{BAD} \\ \texttt{False} \to () \end{pmatrix}$

From Definition 10[p57], for example, we know $\{x \mid x < 10\} \Rightarrow_e \{x \mid x < 12\}$.

The substitution for contracts is defined in Figure 4.5. Here, we assume each bound variable has a unique name.

$$
\begin{array}{lcl}
\{x \mid p\}[e/y] & = & \{x \mid p[e/y]\} \\
(x\colon t_1 \to t_2)[e/y] & = & x\colon t_1[e/y] \to t_2[e/y] \\
(t_1, t_2)[e/y] & = & (t_1[e/y], e_2[e/y]) \\
\texttt{Any}[e/y] & = & \texttt{Any}
\end{array}
$$

Figure 4.5: Substitution for Contracts

4.5.1 Predicate Contract Ordering

We prove that the rule [C-Pred] is sound; that is we prove Theorem 2[p58].

Theorem 2 (Predicate Contract Ordering) *For all expressions p, q, if $p \Rightarrow q$ then $\{x \mid p\} \leqq \{x \mid q\}$.*

Proof We have the following proof for all t_1, t_2, t_3, t_4:

$p \Rightarrow_e q$

$\iff$ (By Definition 10[p57] (Boolean Expression Implication), let

$e_1 = \begin{pmatrix} \texttt{case}\ p\ \texttt{of} \\ \texttt{True} \to () \\ \texttt{False} \to \texttt{BAD} \end{pmatrix}$ and $e_2 = \begin{pmatrix} \texttt{case}\ q\ \texttt{of} \\ \texttt{True} \to () \\ \texttt{False} \to \texttt{BAD} \end{pmatrix}$)

$e_1 \preceq e_2$

$\iff$ (By Definition 7[p48] (Crashes-more-often))

$\forall \mathcal{C}.\ \mathcal{C}[\![e_2]\!] \to^* \texttt{BAD} \Rightarrow \mathcal{C}[\![e_1]\!] \to^* \texttt{BAD}$

$\Rightarrow$ (By (*) below)

$\forall e.\ e$ is crash-free and $(e_1[e/x] \not\to^* \{\texttt{BAD}, \texttt{False}\} \Rightarrow e_2[e/x] \not\to^* \{\texttt{BAD}, \texttt{False}\})$

$\iff$ (By logic and definition of $\in$ in Figure 4.3)

$\forall e.\ e \in \{x \mid e_1\} \Rightarrow e \in \{x \mid e_2\}$

$\iff$ (By Definition 9[p57] (Subcontract))

$\{x \mid e_1\} \leqq \{x \mid e_2\}$

(*) We know $\forall e, a, x.\ e[a/x] \equiv_s \texttt{let}\ x = a\ \texttt{in}\ e$.
Assuming for all crash-free e:
(1) $\forall \mathcal{C}.\ \mathcal{C}[\![e_2]\!] \to^* \texttt{BAD} \Rightarrow \mathcal{C}[\![e_1]\!] \to^* \texttt{BAD}$
(2) $(\texttt{let}\ x = e\ \texttt{in}\ e_1) \not\to^* \{\texttt{BAD}, \texttt{False}\})$
we want to show $(\texttt{let}\ x = e\ \texttt{in}\ e_2) \not\to^* \{\texttt{BAD}, \texttt{False}\}$

Suppose $(\texttt{let}\ x = e\ \texttt{in}\ e_2) \to^* \texttt{BAD}$
By (1), let $\mathcal{C}$ be $\texttt{let}\ x = e\ \texttt{in}\ \bullet$, we have $\mathcal{C}[\![e_1]\!] \to^* \texttt{BAD}$.
That means $(\texttt{let}\ x = e\ \texttt{in}\ e_1) \to^* \texttt{BAD}$.
This contradicts with (2) so our assumption is wrong and we are done.

Suppose $(\texttt{let}\ x = e\ \texttt{in}\ e_2) \to^* \texttt{False}$
By (1), let $\mathcal{C}$ be $\texttt{case}\ (\texttt{let}\ x = e\ \texttt{in}\ \bullet)\ \texttt{of}\ \{\texttt{False} \to \texttt{BAD}\}$, we have $\mathcal{C}[\![e_1]\!] \to^* \texttt{BAD}$.
That means $(\texttt{case}\ (\texttt{let}\ x = e\ \texttt{in}\ e_1)\ \texttt{of}\ \{\texttt{False} \to \texttt{BAD}\}) \to^* \texttt{BAD}$.
That means $(\texttt{let}\ x = e\ \texttt{in}\ e_1) \to^* \{\texttt{BAD}, \texttt{False}\}$.
This contradicts with (2) so our assumption is wrong and we are done.

End of proof. ∎

4.5.2 Dependent Function Contract Ordering

We prove that the rule [C-DepFun] is sound; that is we prove Theorem 3[p59].

Theorem 3 (Dependent Function Contract Ordering) *For all* t_1, t_2, t_3, t_4.

$$\textit{if } t_3 \leqq t_1 \textit{ and } \forall e \in t_3.\ t_2[e/x] \leqq t_4[e/x],\ \textit{then } x\colon t_1 \to t_2 \leqq x\colon t_3 \to t_4$$

Proof We have the following proof for all t_1, t_2, t_3, t_4:

$$\begin{array}{ll} & t_3 \leqq t_1 \text{ and } \forall e_3 \in t_3.\ t_2[e_3/x] \leqq t_4[e_3/x] \\ \iff & \text{(By Definition 9[p57] (Subcontract))} \\ (\dagger_1) & \forall e_1.\ e_1 \in t_3 \Rightarrow e_1 \in t_1 \text{ and } \forall e_3 \in t_3. \forall e_2.\ e_2 \in t_2[e_3/x] \Rightarrow e_2 \in t_4[e_3/x] \\ \Rightarrow & \text{(By the (*) below)} \\ (\dagger_2) & \forall e.\ \forall e_1 \in t_1.\ (e\ e_1) \in t_2[e_1/x] \Rightarrow \forall e_3 \in t_3.\ (e\ e_3) \in t_4[e_3/x] \\ \iff & \text{(By definition of } \in \text{ in Figure 4.3)} \\ & \forall e.\ e \in x\colon t_1 \to t_2 \Rightarrow e \in x\colon t_3 \to t_4 \\ \iff & \text{(By Definition 9[p57] (Subcontract))} \\ & x\colon t_1 \to t_2 \leqq x\colon t_3 \to t_4 \end{array}$$

$(*)$ For all e, assuming:

(1) $\forall e_1.\ e_1 \in t_3 \Rightarrow e_1 \in t_1$ (first clause of the line $\dagger_1$)

(2) $\forall e_3 \in t_3,\ \forall e_2.\ e_2 \in t_2[e_3/x] \Rightarrow e_2 \in t_4[e_3/x]$ (second clause of the line $\dagger_1$)

(3) $\forall e_1 \in t_1.(e\ e_1) \in t_2[e_1/x]$ (LHS of the line $\dagger_2$)

we show $\forall e_3.\ e_3 \in t_3 \Rightarrow (e\ e_3) \in t_4[e_3/x]$as follows.

$$\begin{array}{ll} & e_3 \in t_3 \\ \iff & \text{(By (1))} \\ & e_3 \in t_1 \\ \iff & \text{(By (3))} \\ & (e\ e_3) \in t_2[e_3/x] \\ \iff & \text{(By (2))} \\ & (e\ e_3) \in t_4[e_3/x] \end{array}$$

We are done. ∎

4.5.3 Tuple Contract Ordering

We prove the rule [C-Tup] is sound by showing:

$$\text{For all } t_1, t_2, t_3, t_4.\ \text{if } t_1 \leqq t_3 \text{ and } t_2 \leqq t_4,\ \text{then } (t_1, t_2) \leqq (t_3, t_4)$$

PROOF For all e, if e diverges, then for all t_1, t_2, t_3, t_4, $e \in (t_1, t_2)$ and $e \in (t_3, t_4)$ because a divergent expression satisfies all contracts. By the definition of $\leqq$, we have the desired result $(t_1, t_2) \leqq (t_3, t_4)$. Now, we prove the case when $e \to^* (e_1, e_2)$ as follows.

$$
\begin{array}{ll}
 & t_1 \leqq t_3 \text{ and } t_2 \leqq t_4 \\
\iff & \text{(By Definition 9}^{\text{p57}}\text{ (Subcontract))} \\
 & \forall e_1.\ e_1 \in t_1 \Rightarrow e_1 \in t_3 \text{ and } \forall e_2.\ e_2 \in t_2 \Rightarrow e_2 \in t_4 \\
\iff & \text{(By logic } (\forall x.A) \wedge (\forall y.B) \equiv \forall x, y.\ A \wedge B \text{ if } y \notin fv(A) \text{ and } x \notin fv(B)) \\
 & \forall e_1, e_2.\ e_1 \in t_1 \Rightarrow e_1 \in t_3 \text{ and } e_2 \in t_2 \Rightarrow e_2 \in t_4 \\
\Rightarrow & \text{(By logic } ((A \Rightarrow B) \wedge (C \Rightarrow D)) \Rightarrow ((A \wedge C) \Rightarrow (B \wedge D))) \\
 & \forall e.\ e \to^* (e_1, e_2) \text{ and } ((e_1 \in t_1 \text{ and } e_2 \in t_2) \Rightarrow (e_1 \in t_3 \text{ and } e_2 \in t_4)) \\
\Rightarrow & \text{(By logic } (A \wedge (B \Rightarrow C)) \Rightarrow ((A \wedge B) \Rightarrow (A \wedge C))) \\
 & \forall e.\ (e \to^* (e_1, e_2) \text{ and } e_1 \in t_1 \text{ and } e_2 \in t_2) \\
 & \quad \Rightarrow (e \to^* (e_1, e_2) \text{ and } e_1 \in t_3 \text{ and } e_2 \in t_4) \\
\iff & \text{(By definition of } \in \text{ in Figure 4.3)} \\
 & \forall e.\ e \in (t_1, t_2) \Rightarrow e \in (t_3, t_4) \\
\iff & \text{(By Definition 9}^{\text{p57}}\text{ (Subcontract))} \\
 & (t_1, t_2) \leqq (t_3, t_4)
\end{array}
$$

Note that some tuple contracts are not comparable by $\leqq$, for example: $(\texttt{Ok}, \texttt{Any}) \not\leqq (\texttt{Any}, \texttt{Ok})$ and $(\texttt{Any}, \texttt{Ok}) \not\leqq (\texttt{Ok}, \texttt{Any})$.

4.6 Contract Equivalence

In this section we give formal definition of the equivalence of two contracts. Definition 11[p60] is used in the proof of the Telescoping Property, an important property of contracts in Section 6.2

Definition 11 (Contract Equivalence) *Two closed contracts t_1 and t_2 are equivalent, namely $t_1 \equiv_t t_2$, iff*

$$\forall e.\ e \in t_1 \iff e \in t_2$$

Theorem 4 (Subcontract is Antisymmetric) *For all closed contracts t_1 and t_2, $t_1 \leqq t_2$ and $t_2 \leqq t_1$ iff $t_1 \equiv_t t_2$.*

PROOF

$$
\begin{array}{ll}
 & t_1 \leqq t_2 \text{ and } t_2 \leqq t_1 \\
\iff & \text{(By Definition 9}^{\text{p57}}\text{ (Subcontract))} \\
 & \forall e.\ e \in t_1 \Rightarrow e \in t_2 \text{ and } \forall e.\ e \in t_2 \Rightarrow e \in t_1 \\
\iff & \text{(By logic } (\forall x.\ A(x) \Rightarrow B(x)) \wedge (\forall x.\ B(x) \Rightarrow A(x)) \equiv \forall x.\ A(x) \iff B(x)) \\
 & \forall e.\ e \in t_1 \iff e \in t_2 \\
\iff & \text{(By Definition 11}^{\text{p60}}\text{ (Contract Equivalence))} \\
 & t_1 \equiv_t t_2
\end{array}
$$

End of proof. ∎

For open contracts t, we assume implicitly that there is an environment Δ, which is a mapping from variable to its type, contract and definition (See Definition 8[p57] in Section 4.4).

Lemma 11[p61] and Lemma 12[p61] are needed in proving the main theorem of this thesis. Similar lemmas for tuple contract and `Any` contract can be proved easily; as they are not used anywhere, we omit those two lemmas in this thesis.

Lemma 11 (Predicate Contract Equivalence) *For all expressions e_1 and e_2, if $e_1 \equiv_s e_2$, then $\{x \mid e_1\} \equiv_t \{x \mid e_2\}$.*

Proof We have the following proof:

$$
\begin{array}{ll}
 & e_1 \equiv_s e_2 \\
\iff & \text{(By Theorem 1[p49] (Crashes-more-often is Antisymmetric))} \\
 & e_1 \preceq e_2 \text{ and } e_2 \preceq e_1 \\
\iff & \text{(By Theorem 2[p58] (Predicate Contract Ordering))} \\
 & \{x \mid e_1\} \leqq \{x \mid e_2\} \text{ and } \{x \mid e_2\} \leqq \{x \mid e_1\} \\
\iff & \text{(By Theorem 4[p60] (Subcontract is Antisymmetric))} \\
 & \{x \mid e_1\} \equiv_t \{x \mid e_2\}
\end{array}
$$

End of proof. ∎

Lemma 12[p61] illustrates that, to see two dependent function contracts are equivalent, we require two domains to be equivalent; instead of checking the equivalence of two codomains, we check the equivalence of two ranges. For example:

$$t_1 = x\colon \{x \mid x > 0\} \to \{y \mid y > x\} \qquad t_2 = x\colon \{x \mid x > 0\} \to \{y \mid y > \texttt{abs}\ x\}$$

where function `abs` returns the absolute value of x. In this case, $t_1 \equiv_t t_2$ holds, but $\{y \mid y > x\} \not\equiv_t \{y \mid y > \texttt{abs}\ x\}$.

Lemma 12 (Dependent Function Contract Equivalence) *For all contracts t_1, t_2, t_3, t_4, if $t_1 \equiv_t t_3$ and $\forall e \in t_1.\ t_2[e/x] \equiv_t t_4[e/x]$, then $x\colon t_1 \to t_2 \equiv_t x\colon t_3 \to t_4$.*

Proof We have the following proof.

$$
\begin{array}{ll}
 & t_1 \equiv_t t_3 \text{ and } \forall e \in t_1.\ t_2[e/x] \equiv_t t_4[e/x] \\
\iff & \text{(By Theorem 4[p60] (Subcontract is Antisymmetric))} \\
 & t_1 \leqq t_3 \text{ and } t_3 \leqq t_1 \text{ and} \\
 & (\forall e \in t_1.\ t_2[e/x] \leqq t_4[e/x] \text{ and } \forall e \in t_1.\ t_4[e/x] \leqq t_2[e/x]) \\
\iff & \text{(Since } t_1 \equiv_t t_3, e \in t_1 \iff e \in t_3.) \\
 & t_3 \leqq t_1 \text{ and } \forall e \in t_1.\ t_2[e/x] \leqq t_4[e/x] \text{ and} \\
 & t_1 \leqq t_3 \text{ and } \forall e \in t_1.\ t_4[e/x] \leqq t_2[e/x] \\
\Rightarrow & \text{(By [C-DepFun] in Figure 4.4)} \\
 & x\colon t_1 \to t_2 \leqq x\colon t_3 \to t_4 \text{ and } x\colon t_3 \to t_4 \leqq x\colon t_1 \to t_2 \\
\iff & \text{(By Theorem 4[p60] (Subcontract is Antisymmetric))} \\
 & x\colon t_1 \to t_2 \equiv_t x\colon t_3 \to t_4
\end{array}
$$

We are done. ∎

Chapter 5

Contract Checking

So far we have a nice declarative specification of when a term satisfies a contract. If we could statically check such claims, we would have a powerful tool. For example, if we could show that `main` $\in$ `{x | True}`, then we would have proved that the entire program is crash-free.

In their ground-breaking paper [FF02], Findler & Felleisen describe how to "wrap" a term in a contract-checking wrapper, that checks at run-time

(a) that the term obeys its contract, and

(b) that the context of the term respects the contract.

One route to checking contract satisfaction statically, sketched in §2.3, is to wrap the term in the Findler-Felleisen way, and check that the resulting term cannot crash.

This leads to our main property:

Property 1 (Soundness and Completeness of Contract Checking) *For all expressions e, and crash-free contracts t,*

$$(e \triangleright t) \text{ is crash-free} \quad \Longleftrightarrow \quad e \in t$$

The free variables of e and t must all be bound by the top-level environment Δ, which we leave implicit as before. The form $(e \triangleright t)$ wraps e in a Findler-Felleisen-style contract checker, specified in Figure 5.1. As in the case of contract satisfaction, there are tricky details, as we discuss in Section 5.1. Another subtle but important point is the requirement that the contract t be "crash-free"; this deals with contracts that crash and is discussed in Section 5.3.1.

Property 1^{p63} is very strong. It states that the wrapped term $e \triangleright t$ is crash-free *when and only when* $e \in t$. Certainly, then, if we can prove that $e \triangleright t$ is crash-free, we have proved that $e \in t$. But how can we prove that $e \triangleright t$ is crash-free? There are many ways to do this. Briefly, the approach we take is to perform meaning-preserving transformations on $e \triangleright t$, of precisely the kind that an optimising compiler might perform (inlining, β-reduction, constant folding, etc). If we can "optimise" (i.e. symbolically simplify) the term to a form that is syntactically safe (Section 3.4), then we are done. The better the symbolic simplification, the more contract satisfaction checks will succeed – but none of that affects Property 1^{p63}. For details see Section 7.

$$r \in \{\texttt{BAD}, \texttt{UNR}\} \qquad e \triangleright t \;=\; e \overset{\texttt{BAD}}{\underset{\texttt{UNR}}{\bowtie}} t \qquad\qquad e \triangleleft t \;=\; e \overset{\texttt{UNR}}{\underset{\texttt{BAD}}{\bowtie}} t$$

$$e \overset{r_1}{\underset{r_2}{\bowtie}} \{x \mid p\} \;=\; e \text{ `}\texttt{seq}\text{` } \texttt{case fin } p[e/x] \texttt{ of} \qquad [\texttt{P1}]$$
$$\texttt{True} \to e$$
$$\texttt{False} \to r_1$$

$$e \overset{r_1}{\underset{r_2}{\bowtie}} x\colon t_1 \to t_2 \;=\; e \text{ `}\texttt{seq}\text{` } \lambda v.\ \texttt{let } x = (v \overset{r_2}{\underset{r_1}{\bowtie}} t_1) \qquad [\texttt{P2}]$$
$$\texttt{in } (e\ x) \overset{r_1}{\underset{r_2}{\bowtie}} t_2$$

$$e \overset{r_1}{\underset{r_2}{\bowtie}} (t_1, t_2) \;=\; \texttt{case } e \texttt{ of} \qquad [\texttt{P3}]$$
$$(e_1, e_2) \to (e_1 \overset{r_1}{\underset{r_2}{\bowtie}} t_1,\ e_2 \overset{r_1}{\underset{r_2}{\bowtie}} t_2)$$

$$e \overset{r_1}{\underset{r_2}{\bowtie}} \texttt{Any} \;=\; r_2 \qquad [\texttt{P4}]$$

Figure 5.1: Projection

5.1 Wrappers ▷ and ◁

We have converted the contract satisfaction checking problem to a crash-freeness checking problem. Besides the wrapper ▷, we need to define its dual ◁ as well. That means we define two wrappers, $e \triangleright t$ and $e \triangleleft t$, where e is an expression and t is a contract. These two forms are not part of the syntax of expressions (Figure 3.1); rather they are thought of as macros, which expand to a particular expression. The definition of each expansion is given in Figure 5.1. Informally:

- $e \triangleright t$, pronounced "e `ensures` t", crashes if e does not satisfy t.
- $e \triangleleft t$, pronounced "e `requires` t", crashes if the context does not satisfy t.

Our goal is to define $e \triangleright t$ such that Property 1^{p63} holds.

The expression constructors ▷ and ◁ are dual to each other, so we define $e \triangleright t$ and $e \triangleleft t$ through a combined constructor:

$$e \overset{r_1}{\underset{r_2}{\bowtie}} t$$

which is a term that behaves just like e, except that it throws exception r_1 if e does not respect t, and throws exception r_2 if the wrapped term is used in a way that does not respect t. In the vocabulary of "blame", r_1 means "blame e" while r_2 means "blame the context". Figure 5.1 defines the convenient abbreviations

$$e \triangleright t = e \overset{\texttt{BAD}}{\underset{\texttt{UNR}}{\bowtie}} t \qquad\qquad e \triangleleft t = e \overset{\texttt{UNR}}{\underset{\texttt{BAD}}{\bowtie}} t$$

So $e \triangleright t$ crashes (with `BAD`) if e does not satisfy t, and diverges (with `UNR`) if the context does not respect t. Strictly speaking these two wrappers are more like the wrappers in [BM06] rather than in [FF02]. The detailed differences are discussed in Chapter 11.

The beauty of the constructors lies in [P1] and [P2]. Let us ignore the occurrences of `seq` and `fin` temporarily. The main structure of Figure 5.1 is standard from earlier works [FF02, BM06], and we do not belabour it here. In [P1], we have:

$$\begin{array}{rcl} e \triangleright \{x \mid p\} & = & \texttt{case}\ p[e/x]\ \texttt{of} \\ & & \quad \texttt{True} \to e \\ & & \quad \texttt{False} \to \texttt{BAD} \end{array}$$

$$\begin{array}{rcl} e \triangleleft \{x \mid p\} & = & \texttt{case}\ p[e/x]\ \texttt{of} \\ & & \quad \texttt{True} \to e \\ & & \quad \texttt{False} \to \texttt{UNR} \end{array}$$

The $e \triangleright \{x \mid p\}$ says that if e fails to satisfy the predicate p, it is e's fault because we would like e to **ensure** the property p and we signal this fault with the `BAD`. That means if $e \triangleright \{x \mid p\}$ is crash-free, then the `BAD` is not reachable so we know the predicate p is satisfied. On the other hand, the $e \triangleleft \{x \mid p\}$ says that if e fails to satisfy the predicate p, the rest of the code should be unreachable because we **require** the caller of e to have the property p before e is called. In [P2], we swap the direction of the triangles for the parameter (note the inversion of r_1 and r_2 in the expansion of function contracts):

$$\begin{array}{rcl} e \triangleright t_1 \to t_2 & = & \lambda v.\ ((e\ (v \triangleleft t_1)) \triangleright t_2) \\ e \triangleleft t_1 \to t_2 & = & \lambda v.\ ((e\ (v \triangleright t_1)) \triangleleft t_2) \end{array}$$

The $e \triangleright t_1 \to t_2$ says that in order to ensure the postcondition to be t_2, we require the argument to satisfy the precondition t_1. The $e \triangleleft t_1 \to t_2$ says that if the caller of e (i.e. the argument given to e) cannot ensure t_1, the `BAD` introduced by the $\triangleright$ signals this failure.

It becomes more interesting when we have higher order functions, the direction of the triangle swaps back and forth:

$$\begin{array}{rcl} & & (\lambda x.e) \triangleright (t_1 \to t_2) \to t_3 \\ & = & \lambda v_1.\ (((\lambda x.e)\ (v_1 \triangleleft t_1 \to t_2)) \triangleright t_3) \\ & = & \lambda v_1.\ (((\lambda x.e)\ (\lambda v_2.\ ((v_1\ (v_2 \triangleright t_1)) \triangleleft t_2))) \triangleright t_3) \end{array}$$

Recall the higher-order function `f1 g = (g 1) - 1` in Section 2.1.3, we have:

$$\begin{array}{rcl} & & \texttt{f1} \triangleright (\{x \mid \texttt{True}\} \to \{y \mid y >= 0\}) \to \{r \mid r >= 0\} \\ & = & \ldots \\ & = & \lambda v_1.\ \ \texttt{case}\ (v_1\ 1) >= 0\ \texttt{of} \\ & & \qquad \texttt{True} \to\ \ \texttt{case}\ (v_1\ 1) - 1 >= 0\ \texttt{of} \\ & & \qquad\qquad \texttt{True} \to (v_1\ 1) - 1 \\ & & \qquad\qquad \texttt{False} \to \texttt{BAD} \\ & & \qquad \texttt{False} \to \texttt{UNR} \end{array}$$

As $((v_1\ 1) >= 0)$ does not imply $((v_1\ 1) - 1 >= 0)$, the residual `BAD` indicates a post-condition failure. This illustrates how we get the first error message in Section 2.1.3. As `f1`'s definition does not satisfy its contract, we only use its contract at call sites of `f1`.

As a result, in `f2`, as $(x - 1 \geq 0)$ is not true for all x, we get the second error message in Section 2.1.3.

One thing to note in [P3] is that we expect the expression e to evaluate to a tuple. Recall the example $(\texttt{BAD}, 3) \notin \{x \mid \texttt{snd}\ x > 0\}$, using Property 1[p63] we can verify it by checking crash-freeness of $(\texttt{BAD}, 3) \triangleright \{x \mid \texttt{snd}\ x > 0\}$:

$$
\begin{array}{rl}
 & (\texttt{BAD}, 3)\ \texttt{`seq`}\ \texttt{case snd}\ (\texttt{BAD}, 3) > 0\ \texttt{of} \\
 & \quad \texttt{True} \rightarrow (\texttt{BAD}, 3) \\
 & \quad \texttt{False} \rightarrow \texttt{BAD} \\
\rightarrow^* & \texttt{case snd}\ (\texttt{BAD}, 3) > 0\ \texttt{of} \\
 & \quad \texttt{True} \rightarrow (\texttt{BAD}, 3) \\
 & \quad \texttt{False} \rightarrow \texttt{BAD} \\
\rightarrow^* & (\texttt{BAD}, 3) \\
 & \text{which is not crash-free}
\end{array}
$$

Similarly, we can verify that $(\texttt{BAD}, 3) \in (\texttt{Any}, \{x \mid x > 0\})$ as $\texttt{BAD} \triangleright \texttt{Any} = \texttt{UNR}$ which is crash-free and we also have:

$$
\begin{array}{rl}
 & \texttt{case}\ 3 > 0\ \texttt{of} \\
 & \quad \texttt{True} \rightarrow 3 \\
 & \quad \texttt{False} \rightarrow \texttt{BAD} \\
\rightarrow^* & 3 \\
 & \text{which is crash-free}
\end{array}
$$

The wrapping of `Any` in [P4], while new, is obvious after a moment's thought. For example:

$$
\begin{array}{l}
\texttt{fst} \triangleright (\texttt{Ok}, \texttt{Any}) \rightarrow \texttt{Ok} \\
= \lambda v.((\texttt{fst}\ (v \triangleleft (\texttt{Ok}, \texttt{Any}))) \triangleright \texttt{Ok}) \\
= \lambda v.((\texttt{fst}\ (v \triangleleft (\texttt{Ok}, \texttt{Any})))) \\
= \lambda v.((\texttt{fst}\ (\texttt{case}\ v\ \texttt{of}(a, b) \rightarrow (a \triangleleft \texttt{Ok}, b \triangleleft \texttt{Any})))) \\
= \lambda v.((\texttt{fst}\ (\texttt{case}\ v\ \texttt{of}(a, b) \rightarrow (a, \texttt{BAD}))))
\end{array}
$$

Here we have used the fact that $e \bowtie \texttt{Ok} = e \bowtie \{x \mid \texttt{True}\} = e$. That is, considered as a wrapper `Ok` does nothing at all. In this example we see that the wrapper replaces the second component of the argument to `fst` with `BAD`, so that if `fst` should ever look at it, the program will crash. That is exactly right, because the contract says that the second component can be anything, with contract `Any`.

The polymorphic `Any` contract satisfies Lemma 13[p66], which says that the only contract that makes $\texttt{BAD} \triangleright t$ crash-free is `Any`.

Lemma 13 (Contract `Any` - II) *If* $\texttt{BAD} \triangleright t$ *is crash-free, then* $t = \texttt{Any}$.

Proof By inspecting the definition of $\triangleright$, for all t such that $t \neq \texttt{Any}$, $\texttt{BAD} \triangleright t \rightarrow^* \texttt{BAD}$ which is not crash-free. And we have $\texttt{BAD} \triangleright \texttt{Any} = \texttt{UNR}$ which is crash-free, so we are done. ∎

5.1.1 The use of `seq`

The reader may wonder about the uses of `seq` in [P1] and [P2] of Figure 5.1. The Haskell function `seq` (short for "sequence") is defined as follows:

$$
\begin{array}{rcl}
e_1\ \texttt{`seq`}\ e_2 & = & \texttt{case}\ e_1\ \texttt{of} \\
 & & \quad \texttt{DEFAULT} \rightarrow e_2
\end{array}
$$

which has the following properties:

Lemma 14 (Properties of `seq`)

$$\begin{array}{lrcl} (a) & e_1\!\uparrow & \Rightarrow & (e_1\ \text{`}\texttt{seq}\text{`}\ e_2)\!\uparrow \\ (b) & e_1 \to^* \texttt{BAD} & \Rightarrow & (e_1\ \text{`}\texttt{seq}\text{`}\ e_2) \to^* \texttt{BAD} \\ (c) & e_1 \to^* val \notin \{\texttt{BAD}, \texttt{UNR}\} & \Rightarrow & (e_1\ \text{`}\texttt{seq}\text{`}\ e_2) \equiv_s e_2 \end{array}$$

PROOF By inspecting the operational semantics in Figure 3.3 and definition of `seq`. ∎

It is necessary in the definition of $\bowtie$ to ensure that Property 1^{p63} holds for (a) divergent and (b) crashing terms. For example, if `bot` is a diverging term (defined by `bot = bot`), then Figure 4.3 says that $\texttt{bot} \in \{x \mid \texttt{False}\}$. But if [P1] lacked the `seq`, we would have

$$\begin{array}{l} \texttt{bot} \triangleright \{x \mid \texttt{False}\} \\ = \texttt{case False of \{ True -> bot; False -> BAD \}} \\ = \texttt{BAD}, \quad \text{which is not crash-free} \end{array}$$

thus contradicting Property 1^{p63}.

Dually, we must ensure that $\texttt{BAD} \notin \texttt{Ok} \to \texttt{Any}$. Without the `seq` in [P2] we would get

$$\begin{array}{rl} \texttt{BAD} \triangleright \texttt{Ok} \to \texttt{Any} & = \lambda v.\ ((\texttt{BAD}\ (v \triangleleft \texttt{Ok})) \triangleright \texttt{Any}) \\ & = \lambda v.\ \texttt{UNR}, \quad \text{which is crash-free} \end{array}$$

again contradicting Property 1^{p63}. This also means

$$\texttt{BAD} \not\equiv_s \lambda x.\texttt{BAD}$$

because `BAD` denotes a crash while $\lambda x.\texttt{BAD}$ is a value which will only crash when applied to an argument.

Have we covered all the cases? A quick check shows that:

- If $e \to^* \texttt{BAD}$, the e ‘`seq`‘ prevents e from satisfying the contract unless the contract is `Any`.
- If $e\!\uparrow$, then $e \triangleright t$ satisfies all contracts.

More solidly, Property 1^{p63} goes through with the definitions of Figure 5.1.

5.1.2 Aside: Why Only Crash-free Terms Satisfy Predicate Contracts

In Section 4.3.1 we promised to explain why we chose to allow only *crash-free* terms to satisfy a predicate contract, regardless of the predicate. An obvious alternative design choice for contract satisfaction would be to drop the “e is crash-free” condition in the predicate contract case:

$$e \in \{x \mid p\} \iff e\!\uparrow \text{ or } p[e/x] \not\to^* \{\texttt{BAD}, \texttt{False}\} \qquad [\text{B1}]$$

Then we could get rid of `Any`, because `{x | True}` would do instead. On the other hand, a polymorphic contract meaning "crash-free" is extremely useful in practice, so we would probably need a new contract `Ok` (now not an abbreviation) defined thus:

$$e \in \texttt{Ok} \iff e \text{ is crash-free} \qquad \text{[B2]}$$

This all seems quite plausible, but it has a fatal flaw: *we could not find a definition for* $\triangleright$ *that validates our main theorem.* That is, our chosen definition for $\in$ makes Figure 5.1 work out, whereas the otherwise-plausible alternative appears to prevent it doing so.

Suppose we have [B1] instead of [A1], that means $(\texttt{BAD}, \texttt{BAD}) \in \{x \mid \texttt{True}\}$. However, according to [P1], we have $(\texttt{BAD}, \texttt{BAD}) \triangleright \{x \mid \texttt{True}\} = (\texttt{BAD}, \texttt{BAD})$ which is not crash-free. This means Property 1^{p63} fails. Can we change [P1] to fix the theorem? It is hard to see how to do so. The revised rule must presumably look something like

$$e \triangleright \{x \mid p\} = \texttt{case fin } p[e/x] \texttt{ of \{ True} \to ???; \texttt{ False} \to \texttt{BAD \}}$$

But what can we put for "???"? Since $e \triangleright t$ is supposed to behave like e if $p[e/x]$ holds, the "???" must be e — but then $\texttt{BAD} \triangleright \{x \mid \texttt{True}\}$ would not be crash free. This difficulty motivates our choice that predicate contracts are satisfied only by crash-free terms.

5.2 Contracts that Diverge

Our system allows non-termination both in the *programs* we verify, and in their *specifications* (contracts), which is most unusual for a system supporting static verification.

As we discussed in Section 4.3.4, we allow non-termination for *programs* because we work with a real-life programming language, in which many functions actually do not terminate. We do not want to exclude non-termination in general, even for specifications, because we do not want to be forced to perform termination proofs, which are often tedious to do. Since the current advances in automatic termination proofs are still limited, especially for lazy programs, requiring termination would put a substantial extra burden on the user of our system.

What about divergent *contracts*? Many program verification systems for functional programming, such as HOL, systems based on dependent types (Coq, Agda), and ACL2, do not allow *any* non-terminating definitions. The main reason is that this introduces an immediate unsoundness in these systems. For example, by an (unsound) induction proof, a constant defined as `let x = x` could be proved equal to both `1` and `2`, concluding that `1=2`.

But it would be an onerous burden to insist that all contracts terminate, because the programmer can write arbitrary Haskell in contracts, and proving termination of arbitrary Haskell programs is hard. Furthermore, allowing non-termination in specifications is of direct benefit. Consider a function `zipE` which requires two inputs to have the same length:

```
{-# CONTRACT zipE :: xs:Ok -> {ys | sameLen xs ys} -> Ok #-}
```

```
zipE []     []     = []
zipE (x:xs) (y:ys) = (x,y) : zipE xs ys
zipE _      _      = error "unequal lengths"

sameLen []     []     = True
sameLen (x:xs) (y:ys) = sameLen xs ys
sameLen _      _      = False
```

Here, two infinite lists satisfy the contract for `ys`, since `(sameLen xs ys)` diverges, and indeed `zipE` does not crash for such arguments. Care is necessary in writing the contract: if we had instead said `length xs == length ys`, the contract would diverge if only *one* argument was infinite, but `zipE` would crash for such arguments, so it would not satisfy this alternative contract. In this way, *safety properties* over infinite structures are allowed. (Safety properties are properties that always have finite counter-examples whenever there exists any counter-example.)

Why is our approach sound? First, any contract we verify for a program only deals with partial correctness. In other words, all contracts are inhibited by non-terminating programs as well. Second, our system does not include reasoning mechanisms like equational reasoning. In fact, our system has no means of expressing equality at all! Everything is expressed in terms of Haskell expressions evaluating to boolean values, crashing, or not-terminating. Third, any specification that does not terminate is semantically the same as a `True` contract. Why? Because of the mysterious `fin` construct, as we discuss next.

5.2.1 Using `fin` in Contract Wrappers

Suppose we have the top-level definition `bot = bot`; that is, `bot` diverges. Now consider $e =$ `(BAD,BAD)` and $t =$ `{x | bot}`. Then $e \notin t$ (since e is not crash-free). If we did not use `fin` in the definition of $\bowtie$ (Figure 5.1), $e \triangleright t$ would reduce to this term:

```
case bot of { True -> (BAD, BAD); False -> BAD }
```

This term is contextually equivalent to `bot` itself, and so it is crash-free, contradicting Property 1[p63].

What to do? Execution has gotten stuck evaluating the diverging contract, and has thereby missed crashes in the term itself. Our solution is to limit the work that can be spent on contract evaluation. The *actual* definition of $\bowtie$ makes $e \triangleright t$ equal to

```
case (fin_0 bot) of
  True  -> (BAD, BAD)
  False -> BAD
```

The operational semantics of `fin` (Figure 3.3) gives a finite M units of "fuel" to each `fin`. Each reduction under a `fin` increases the subscript on the `fin` until it reaches the maximum M (rule [E-fin1]). When the `fin` subscript n reaches the limit M, `fin` gives up

and returns `True` (rule [E-fin3]). For example: $e =$ `if fin (False || False) then` 1 `else` 2. Then we have:

$$\begin{array}{lll} e & \rightarrow_0^* & 1 \quad (M = 0 \text{ means no fuel}) \\ e & \rightarrow_1^* & 1 \end{array}$$

but

$$\begin{array}{l} e \rightarrow_2^* 2 \\ e \rightarrow_3^* 2 \\ e \rightarrow_4^* 2 \end{array}$$

Now, recall the example at the beginning of Section 5.2.1, *for any finite* M, we have

$$e \triangleright t \ \rightarrow_M^* \ \texttt{(BAD,BAD)}$$

So we define our full-scale reduction relation $\rightarrow^*$ in terms of $\rightarrow_M^*$:

Definition 12 (Reduction) *We say that* $e \rightarrow^* val$ *iff there exists* N *such that for any* $M \geq N$ *we have* $e \rightarrow_M^* val$.

Under this definition, $e \triangleright t \rightarrow^*$ `(BAD,BAD)`, and Property 1^{p63} holds.

5.2.2 Practical Consequences

This may all seem a bit complicated or artificial, but it is very straightforward to implement. First, remember that we are concerned with static verification, not dynamic checking. Uses of `fin` are introduced only to check contract satisfaction, and are never executed in the running program. Second, our technique to check that $e \triangleright t$ is crash-free is to optimise it and check for syntactic safety. To be faithful to the $\rightarrow^*$ semantics, we need only *refrain* from "optimising" `(case (fin bot) of <alts>)` to `bot`, thereby retaining any `BAD`s lurking in `<alts>`. Since this particular optimisation is a tricky one anyway, it is quite easy to omit! In other words, in our static contract checking, we can safely omit `fin` and the rules [E-fin1,2,3].

5.2.3 Summary

By being careful with our definition of $\rightarrow^*$, we can retain Property 1^{p63} in its full, bi-directional form. This approach is, of course, only available to us because we are taking a static approach to verification. A dynamic checker cannot avoid divergence in contracts (since it must evaluate them), and hence must lose the ($\Rightarrow$) direction of Property 1^{p63}, as indeed is the case in [BM06].

5.3 Contracts that Crash

Our goal is to detect crashes in a *program* with the help of contracts, we do not expect contracts themselves to introduce crashes. One approach, taken by Blume & McAllester

[BM06], is to prohibit a contract from mentioning any function that might crash. But that is an onerous restriction, as we argued in Section 2.1.7.

It is attractive simply to allow arbitrary crashes in contracts; after all, Figure 4.3 specifies exactly which terms inhabit even crashing contracts. Alas, if we drop the (still-to-be-defined) condition "crash-free contract" from Property 1^{p63}, the ($\Rightarrow$) direction still holds, but the ($\Leftarrow$) direction fails. Here is a counter-example involving a crashing contract. We know that:

$$\lambda x.x \in \{x \mid \texttt{BAD}\} \to \texttt{Ok}$$

because the only expression that satisfies $\{x \mid \texttt{BAD}\}$ is an expression that diverges and a diverging expression satisfies `Ok`. But we have:

$$\begin{aligned}\lambda x.x \triangleright \{x \mid \texttt{BAD}\} \to \texttt{Ok} &= \lambda v.(\lambda x.x\ (v \triangleleft \{x \mid \texttt{BAD}\})) \\ &= \lambda v.(v \triangleleft \{x \mid \texttt{BAD}\}) \\ &= \lambda v.(v\ \text{`}\texttt{seq}\text{`}\ \texttt{BAD}) \\ &\quad \text{which is not crash-free}\end{aligned}$$

5.3.1 Crash-free Contracts

So unrestricted crashes in contracts invalidates (one direction of) Property 1^{p63}. But no one is asking for unrestricted crashes! For example, this contract does not make much sense:

$$t_{bad} \;=\; xs : \texttt{Ok} \to \{r \mid r > \texttt{head}\ xs\}$$

What does it mean if the argument list is empty? Much more plausible is a contract like this (see §2.1.7):

$$t_{good} \;=\; xs : \{xs \mid \texttt{not}\ (\texttt{null}\ xs)\} \to \{r \mid r > \texttt{head}\ xs\}$$

which specifies that the argument list is non-empty, and guarantees to return a result bigger than head of the argument. You might wonder what happens if the first element of the list xs is `BAD`, that means $\texttt{head}\ xs \to^* \texttt{BAD}$ even if the list xs is non-empty. This will not occur because only a crash-free expression satisfies a predicate contract (recalling [A1] in Figure 4.3). Since xs satisfies a predicate contract $\{xs \mid \texttt{not}\ (\texttt{null}\ xs)\}$, it does not contain crashing elements.

Thus motivated, we define a notation of a "crash-free" contract:

Definition 13 (crash-freeness) *A contract t is crash-free iff*

t is $\{x \mid p\}$ and p is crash-free
or t is $x\colon t_1 \to t_2$ and t_1 is crash-free and for all $e_1 \in t_1, t_2[e_1/x]$ is crash-free
or t is (t_1, t_2) and both t_1 and t_2 are crash-free
or t is `Any`

This definition is essentially the same as that of T_{safe} in [BM06], although perhaps a little more straightforward. It simply asks that the predicates in a contract are crash-free *under the assumption that the dependent function arguments satisfy their contracts.* So, under

this definition, t_{bad} is ill-formed while t_{good} is crash-free. The latter is crash-free because `head` xs is crash-free for every xs that satisfies `not` (`null` xs).

For another example:

```
{-# CONTRACT f :: k:({x | x > 0} -> Ok) -> {r | k (-3) == r} #-}
```

is not crash-free because for a function satisfying `{x | x > 0} -> Ok`, the call `k (-3)` may crash. On the other hand, the contract

```
{-# CONTRACT  f :: k:({x | x > 0} -> Ok) -> {r | k 3 == r} #-}
```

is crash-free.

5.3.2 Wrapping Dependent Function Contracts

Recall [P2] from Figure 5.1:

$$e \overset{r_1}{\underset{r_2}{\bowtie}} x\colon t_1 \to t_2 = e \text{ `seq` } \lambda v.\ ((e\ (v \overset{r_2}{\underset{r_1}{\bowtie}} t_1)) \overset{r_1}{\underset{r_2}{\bowtie}} t_2[(v \overset{r_2}{\underset{r_1}{\bowtie}} t_1)/x])$$

Notice that v is wrapped by $v \overset{r_2}{\underset{r_1}{\bowtie}} t_1$ even in the contract t_2, as well as in the argument to e. Could we simplify [P2] by omitting this wrapping, thus?

$$e \overset{r_1}{\underset{r_2}{\bowtie}} x\colon t_1 \to t_2 = e \text{ `seq` } \lambda v.\ ((e\ (v \overset{r_2}{\underset{r_1}{\bowtie}} t_1)) \overset{r_1}{\underset{r_2}{\bowtie}} t_2[v/x])$$

No, we could not: Property 1^{p63} would fail again. Here is a counter-example.

```
{-# CONTRACT h :: {x | not (null x)} -> {r | head x == r} #-}
h (y:ys) = y
```

Now `h` satisfies its contract t_h, but $\texttt{h} \triangleright t_h$ is not crash-free, as the reader may verify.

We remarked earlier that Blume & McAllester require that contracts only call crash-free functions. But the wrapping of v inside t_2 in rule [P2] might *itself* introduce crashes, at least if t_2 uses x in a way that does not respect t_1. They therefore use another variant of [P2], as follows:

$$e \overset{r_1}{\underset{r_2}{\bowtie}} x\colon t_1 \to t_2 = e \text{ `seq` } \lambda v.\ ((e\ (v \overset{r_2}{\underset{r_1}{\bowtie}} t_1)) \overset{r_1}{\underset{r_2}{\bowtie}} t_2[(v \overset{r_2}{\underset{\texttt{UNR}}{\bowtie}} t_1)/x])$$

Notice the "`UNR`" introduced out of thin air in the wrapping of v in t_2, which is enough to maintain their no-crashing invariant. Happily, if the contracts are crash-free (which we need anyway, so that it is possible to call `head`) there is no need for this somewhat ad-hoc fix.

5.3.3 Practical Consequence

One might worry that the crash-freeness condition in Property 1^{p63} makes the verification task more onerous: perhaps to prove $e \in t$ now we must check two things (a) that t is well formed and (b) that $e \triangleright t$ is crash-free. Happily, this is not necessary, because the ($\Rightarrow$) of Property 1^{p63} holds for arbitrary t:

Theorem 5 (One Direction of Grand Theorem) *For all closed expression e, for all contract t,*

$$(e \triangleright t) \textit{ is crash-free} \;\Rightarrow\; e \in t$$

PROOF The proof of Theorem 5 is the same as the proof for the direction ($\Rightarrow$) of Property 1^{p63} (i.e. Theorem 9^{p81}) because only the proof for the direction ($\Leftarrow$) of Property 1^{p63} requires the condition that t to be crash-free.

We inspect the proof for Theorem 9^{p81} in Section 6.1 case by case. For the case $e \rightarrow^* \texttt{BAD}$, examining the direction ($\Rightarrow$), Lemma 4^{p46} (preservation of crash-freeness) and Lemma 13^{p66} (Contract `Any` - II) are called, but none of them require crash-free t. For the case $t = x\colon t_1 \rightarrow t_2$, Lemma 14^{p67}(c) (properties of `seq`), Lemma 5^{p46} (crash-free function), Theorem 11^{p94} (projection pair), Theorem 10^{p93} (congruence of $\preceq$), Lemma 9^{p49} (c) (property of $\preceq$)) are called, but none of them require crash-free t. It is obvious that the rest of the cases do not require crash-free t, either.

5.3.4 Aside: Conjecture for Ill-formed Contracts

In Section 5.3.1, we show that one direction of Property 1^{p63} does not hold for ill-formed contracts; that is, it is *not* the case that:

$$e \in t \quad \Rightarrow \quad (e \triangleright t) \text{ is crash-free}$$

where we lift the condition that t is crash-free. However, to make Property 1^{p63} hold for both directions, for arbitrary contracts (including ill-formed ones), we may play the following trick. Previously, we introduced a special function `fin` that converts divergence to `True` and untouches other expressions. That means `(fin BAD)` evaluates to `BAD`. Suppose we enhance the definition of `fin` by making it convert a crash to `False`. That is:

$$\texttt{fin BAD} = \texttt{False}$$

Recall the counter example given in Section 5.3.1:

$$\lambda x.x \in \{x \mid \texttt{BAD}\} \rightarrow \{r \mid \texttt{True}\}$$

With this enhanced definition of `fin`, We now have:

$$
\begin{array}{ll}
 & \lambda x.x \triangleright \{x \mid \texttt{BAD}\} \to \{r \mid \texttt{True}\} \\
= & \text{(By definition of } \triangleright \text{ and } \triangleleft\text{)} \\
 & \lambda v.(\lambda x.x\ (v \triangleleft \{x \mid \texttt{BAD}\})) \\
= & \text{(By } \beta\text{-reduction)} \\
 & \lambda v.(v \triangleleft \{x \mid \texttt{BAD}\}) \\
= & \text{(By definition of } \triangleleft\text{)} \\
 & \lambda v.(v\ \text{`}\texttt{seq}\text{`}\ (\texttt{case fin BAD of } \{\texttt{True} \to v;\ \texttt{False} \to \texttt{UNR}\})) \\
= & \text{(Since } \texttt{fin BAD} = \texttt{False}\text{)} \\
 & \lambda v.(v\ \text{`}\texttt{seq}\text{`}\ \texttt{UNR}),\ \text{which is crash-free}
\end{array}
$$

This leads us to the following conjecture, which re-states Property 1^{p63} without the requirement for the contract t to be crash-free.

Conjecture 1 (Sound and Complete for All Contracts) *If* `fin BAD = False`*, then for all closed* e*, (possibly open)* t*,*

$$e \in t \quad \Longleftrightarrow \quad (e \triangleright t)\ \textit{is crash-free}$$

However, we have been unable either to prove this conjecture, or to find a counter example! The proof for Property 1^{p63} fails for the following reason.

With the enhanced definition of `fin` (i.e. `fin` converting `BAD` to `False`), Theorem 10^{p93} (Congruence of $\preceq$) in Section 6.5 fails. Theorem 10^{p93} states that:

$$\forall e_1, e_2.\quad e_1 \preceq e_2 \quad \Longleftrightarrow \quad \forall \mathcal{C},\ \ \mathcal{C}[\![e_1]\!] \preceq \mathcal{C}[\![e_2]\!]$$

Here is a counter example:

Let $e_1 = \texttt{BAD}$ and $e_2 = \texttt{True}$.
By Definition 7^{p48} (Crashes-More-Often), we know $e_1 \preceq e_2$.
We want to show that there exists a context $\mathcal{C}$, such that $\mathcal{C}[\![e_1]\!] \not\preceq \mathcal{C}[\![e_2]\!]$.
Here is such a context: $\mathcal{C} =$ `case fin` $[\![\bullet]\!]$ `of`
`True` $\to$ `BAD`
`False` $\to 5$
where $\mathcal{C}[\![e_1]\!] = 5 \not\preceq \texttt{BAD} = \mathcal{C}[\![e_2]\!]$

We see that a context with `fin` gives us problem. That means the congruence theorem for $\preceq$ holds for all context $\mathcal{C}$, such that `fin` $\notin_s \mathcal{C}$ (i.e. `fin` not syntactically occurring in $\mathcal{C}$). Inspecting the places where Theorem 10^{p93} is called in proving Property 1^{p63}, we do have `fin` in the context. So we get stuck.

5.4 Recursion

You might wonder whether the wrappers $\triangleright$ and $\triangleleft$ work for recursive functions. Recall the idea of treating free variables as parameters in Section 4.4, we can apply it to recursive

functions as well. For example, we have:

$$f \in t$$
$$f = \ldots f \ldots$$

To simplify our presentation, we assume we do not have other top-level function calls and only one self-recursive call in the definition of f. We treat the f in recursive calls as a free variable. So instead of checking $f \in t$, we check $\lambda f.\ \ldots f \ldots \in t \to t$. That means we check whether $(\lambda f.\ \ldots f \ldots) \triangleright t \to t$ is crash-free or not. If we elaborate this term, we get:

$$\begin{array}{ll} & (\lambda f.\ \ldots f \ldots) \triangleright t \to t \\ = & \text{(By definition of } \triangleright\text{)} \\ & \lambda v_1.\ (((\lambda f.\ \ldots f \ldots)\ (v_1 \triangleleft t)) \triangleright t) \\ \to & \text{(By } \beta\text{-reduction)} \\ & \lambda v_1.\ (((\ldots f \ldots)[(v_1 \triangleleft t)/f]) \triangleright t) \end{array}$$

If we apply the last line to f, we get:

$$((\ldots f \ldots)[(f \triangleleft t)/f]) \triangleright t$$

That means we replace each f in recursive calls by $f \triangleleft t$. Suppose $t = t_1 \to t_2$, we have:

$$\begin{array}{ll} & f \triangleright t_1 \to t_2 \\ = & (\ldots (f \triangleleft t_1 \to t_2) \ldots) \triangleright t_1 \to t_2 \\ = & \text{(By definition of } \triangleright \text{ and } \triangleleft\text{)} \\ (*) & \lambda v_2.\ ((\ldots (\lambda v_1.\ ((f\ (v_1 \triangleright t_1)) \triangleleft t_2))\ \underline{(v_2 \triangleleft t_1)} \ldots) \triangleright t_2) \end{array}$$

Recall that the symbol $\triangleright$ means "ensures" (i.e. "check") while the symbol $\triangleleft$ means "requires" (i.e. "assume"). Basically, the line of the above derivation indicated by $(*)$ says that:

- we assume the precondition holds (indicated by $\underline{(v_2 \triangleleft t_1)}$) at the entry of the definition body, we check whether the precondition holds (indicated by $(v_1 \triangleright t_1)$) before the entry of each recursive call.
- we assume the postcondition holds for each recursive call (indicated by $(f\ (v_1 \triangleright t_1)) \underline{\triangleleft\ t_2}$), and we check whether the postcondition holds for the whole function body (indicated by the $\triangleright\ t_2$ at the end of the expression).

This is the same idea as the Hoare logic rule for a procedure call.

5.5 Contract Properties

In this section, we first give a list of interesting properties that contracts enjoy and then discuss some of them in detail.

5.5.1 Properties Overview

The $\preceq$ relation, the satisfaction $\in$ and the two constructors $\triangleright$ and $\triangleleft$ enjoy many nice properties shown in Figure 5.2. These lemmas form a basis for proving our main result: Property 1[p63]. Property 1[p63] is similar to the soundness and completeness theorems in [BM06]. We give a complete proof of the theorem as well as these lemmas in Chapter 6. The proof technique is different from that in [BM06].

Lemma 15[p86] (Telescoping Property)	For all e, crash-free t. $(e \underset{r_2}{\overset{r_1}{\bowtie}} t) \underset{r_4}{\overset{r_3}{\bowtie}} t = e \underset{r_4}{\overset{r_1}{\bowtie}} t$
Lemma 19[p92] (Key Lemma)	For all crash-free e, crash-free t, $e \triangleleft t \in t$.
Lemma 20[p95] (Idempotence)	(a) $e \triangleright t \triangleright t \equiv e \triangleright t$ (b) $e \triangleleft t \triangleleft t \equiv e \triangleleft t$
Lemma 21[p96] (Conditional Projection) (w.r.t. $\preceq$,$\succeq$)	For all e, crash-free t, if $e \in t$, then (a) $e \triangleleft t \preceq e$ (b) $e \triangleright t \succeq e$
Theorem 7[p79] (Monotonicity of $\in$)	If $e_1 \in t_1$ and $e_1 \preceq e_2$, then $e_2 \in t$
Theorem 10[p93] (Congruence)	$\forall e_1, e_2.\ e_1 \preceq e_2 \iff \forall \mathcal{C},\ \mathcal{C}[\![e_1]\!] \preceq \mathcal{C}[\![e_2]\!]$
Theorem 11[p94] (Projection Pair)	$\forall e \in t.\ e \triangleright t \triangleleft t \preceq e$
Theorem 12[p94] (Closure Pair)	$\forall e \in t.\ e \preceq e \triangleleft t \triangleright t$

Figure 5.2: Properties of $\triangleright$ and $\triangleleft$

5.5.2 Contracts are Projections

In [FB06], Findler & Blume discovered that contracts (a re-functionalized contract implementation based on Findler-Felleisen's dynamic contract checking algorithm [FF02]) are pairs of projections (similar, but not the same, idea to our $\triangleright$ and $\triangleleft$). That means given a contract t, $\lambda e.\mathcal{W}_t(e)$ is a projection where $\mathcal{W}_t$ is a wrapper function. To be a projection w.r.t. a partial ordering $\sqsubseteq$, a function p must satisfy these two properties:

1. $p \circ p = p$ (idempotence)
2. $p \sqsubseteq 1$ (result of projection contains no more information than its input)

In Findler & Blume's words:

> The first property means that it suffices to apply a contract once; the second property means that a contract cannot add behaviour to a value. That is, the contract may replace some parts of its input with errors, but it must not change any other behaviour. Technically, Scott's projections [Sco76] only add $\bot$ (divergence), but errors are a better match for our [FB06] work.

Blume & McAllester [BM06] have two types of exceptions $\bot$ (divergence) and $\top$ (error) (where each $\top$ is labelled with i, i.e. $\top_i$, to distinguish the crashes). However, in Findler & Felleisen's work [FF02] and the follow-up work by Findler & Blume [FB06], there

is only one exception `error` (`blame` in [FB06]). That is why in the above quote, they clarify that their contract wrapper may replace some parts of its input with errors, but it does not change any other behaviour. Note that there is no similar claim by Blume & McAllester [BM06].

Our work is closer to Blume & McAllester [BM06] as we have two exceptional values `BAD` (error) and `UNR` (divergence). Are our wrappers projections?

Based on Definition 6[p47] (Behaves-the-same), our contract wrapper $\bowtie$ is a projection with the partial ordering $\ll_{\{r_1,r_2\}}$ (where $r_1, r_2 \in \{\texttt{BAD}, \texttt{UNR}\}$) on terms:

$$(1)\ e \underset{r_2}{\overset{r_1}{\bowtie}} t \underset{r_2}{\overset{r_1}{\bowtie}} t = e \underset{r_2}{\overset{r_1}{\bowtie}} t \qquad\qquad (2)\ e \underset{r_2}{\overset{r_1}{\bowtie}} t \ll_{\{r_1,r_2\}} e$$

The (1) and (2) are Lemma 20[p95] (Idempotence) and Lemma 23[p97] (Behaviour of Projection) whose proof can be found in Section 6.7 and Section 6.8 respectively.

In Figure 5.2, note that the projections $\triangleleft$ and $\triangleright$ are only projections (w.r.t. $\preceq$,$\succeq$) if $e \in t$. If we drop the condition, a counter-example for $e \triangleleft t \preceq e$ is:

$$\begin{array}{ll} & \lambda x.x \triangleleft \{x \mid \texttt{True}\} \to \{r \mid \texttt{False}\} \\ = & (\text{By } \beta\text{-reduction and the fact } e \bowtie \{r \mid \texttt{True}\} = e) \\ & \lambda v.\ ((\lambda x.x\ v) \triangleleft \{r \mid \texttt{False}\}) \\ = & \lambda v.\ (v\ \text{`}\texttt{seq}\text{`}\ \texttt{case fin False of}\ \{\texttt{True} \to v; \texttt{False} \to \texttt{UNR}\}) \\ = & \lambda v.\ (v\ \text{`}\texttt{seq}\text{`}\ \texttt{UNR}) \\ \not\preceq & \lambda x.x \end{array}$$

In this counter example, $\lambda x.x \notin \{x \mid \texttt{True}\} \to \{r \mid \texttt{False}\}$ because $5 \in \{x \mid \texttt{True}\}$ while $((\lambda x.x)\ 5) \notin \{r \mid \texttt{False}\}$ (only a divergent expression satisfies $\{r \mid \texttt{False}\}$). Then is it possible that $e \preceq e \triangleleft t$ (i.e. $e \triangleleft t \succeq e$)? Well, a counter example for $e \triangleleft t \succeq e$ is:

$$\begin{array}{ll} & \lambda x.x \triangleleft \{x \mid \texttt{False}\} \to \{r \mid \texttt{True}\} \\ = & (\text{By } \beta\text{-reduction and the fact } e \bowtie \{r \mid \texttt{True}\} = e) \\ & \lambda v.\ (\lambda x.x\ (v \triangleright \{r \mid \texttt{False}\})) \\ = & \lambda v.\ (v\ \text{`}\texttt{seq}\text{`}\ \texttt{case fin False of}\ \{\texttt{True} \to v; \texttt{False} \to \texttt{BAD}\}) \\ = & \lambda v.\ (v\ \text{`}\texttt{seq}\text{`}\ \texttt{BAD}) \\ \not\succeq & \lambda x.x \end{array}$$

We now examine the other projection $e \triangleright t \succeq e$ (i.e. $e \preceq e \triangleright t$). A counter example for $e \triangleright t \succeq e$ is:

$$\begin{array}{ll} & \lambda x.x \triangleright \{x \mid \texttt{True}\} \to \{r \mid \texttt{False}\} \\ = & (\text{By } \beta\text{-reduction and the fact } e \bowtie \{r \mid \texttt{True}\} = e) \\ & \lambda v.\ ((\lambda x.x\ v) \triangleright \{r \mid \texttt{False}\}) \\ = & \lambda v.\ (v\ \text{`}\texttt{seq}\text{`}\ \texttt{case fin False of}\ \{\texttt{True} \to v; \texttt{False} \to \texttt{BAD}\}) \\ = & \lambda v.\ (v\ \text{`}\texttt{seq}\text{`}\ \texttt{BAD}) \\ \not\succeq & \lambda x.x \end{array}$$

Again, $\lambda x.x \notin \{x \mid \texttt{True}\} \to \{r \mid \texttt{False}\}$. Is it possible to have $e \triangleright t \preceq e$? Again, no. A counter example for $e \triangleright t \preceq e$ is:

$$\begin{array}{ll} & \lambda x.x \triangleright \{x \mid \texttt{False}\} \to \{r \mid \texttt{True}\} \\ = & \text{(By } \beta\text{-reduction and the fact } e \bowtie \{r \mid \texttt{True}\} = e) \\ & \lambda v.\ (\lambda x.x\ (v \triangleleft \{r \mid \texttt{False}\})) \\ = & \lambda v.\ (v\ \text{`}\texttt{seq}\text{`}\ \texttt{case fin False of}\ \{\texttt{True} \to v; \texttt{False} \to \texttt{UNR}\}) \\ = & \lambda v.\ (v\ \text{`}\texttt{seq}\text{`}\ \texttt{UNR}) \\ \not\preceq & \lambda x.x \end{array}$$

Why do Findler et al claim that a contract wrapper is unconditionally a projection in [FF02, FB06] and here we need a condition $e \in t$? As mentioned at the beginning of Section 5.5, in [FF02, FB06], only one exceptional value (`error`) is used. That means, instead of having two exceptional values `BAD` and `UNR`, they have only `BAD`, but give each `BAD` a label *lbl*. For example, (`BAD` "server") indicates that the function definition itself should be blamed as (assuming the precondition holds) the function does not produce a result that satisfies the required postcondition. For another example, (`BAD` "client") indicates that the caller does not supply an argument that meets the callee's precondition. That is why an expression wrapped with their contract wrapper is a projection – either behaving the same as the original expression or throwing an exception (`BAD` *lbl*). This single exception design makes sense because they do contract checking at *run-time*. Once a witness (a run-time data that triggers the `BAD` *lbl*) is found, the execution stops and the *lbl* (which may consist of a function name) is blamed. In our case, we do contract checking at *compile-time* in a *modular* fashion:

- At each function definition, we check whether a function meets its postcondition by assuming the precondition holds. This assumption is indicated by `UNR`. That means if the precondition does not hold, we get `UNR` which is crash-free so that the post-condition checking is not disturbed. (Reminder: we convert contract satisfaction checking to crash-freeness checking)
- At each call site, we check whether an argument given to a function being called satisfies the function's precondition. If it does, we know that the function will produce a result that satisfies the agreed contract. Again, we do not want any postcondition failure to disturb our precondition checking, so if postcondition fails, we get `UNR`.

This also explains why the use of `UNR` is important in our framework.

5.5.3 Contracts Ordering w.r.t. Crashes-more-often

We now try to order wrapped expressions with the crashes-more-often operator. We have not found any usefulness of Theorem 6[p78] yet, but it corresponds to the Theorem 4 and the Theorem 5 (2) in [FB06].

Theorem 6 (Subcontract and Crashes-more-often Ordering) *For all t_1 and t_2,*

$$\forall e.\ e \triangleright t_1 \preceq e \triangleright t_2 \quad \Rightarrow \quad t_1 \leqq t_2$$

PROOF We have the following proof:

$$\begin{array}{ll} & \forall e.\ e \triangleright t_1 \preceq e \triangleright t_2 \\ \Rightarrow & \text{(By Lemma } 9^{\text{p49}} \text{ (c) (Properties of Crashes-more-often - II))} \\ & \forall e.\ e \triangleright t_1 \text{ is crash-free} \Rightarrow e \triangleright t_2 \text{ is crash-free} \\ \Rightarrow & \text{(By Theorem } 9^{\text{p81}} \text{ (grand theorem))} \\ & \forall e.\ e \in t_1 \Rightarrow e \in t_2 \\ \Longleftrightarrow & \text{(By Definition } 9^{\text{p57}} \text{ (Subcontract))} \\ & t_1 \leqq t_2 \end{array}$$

5.5.4 Monotonicity of Satisfaction

Theorem 7^{p79} says that if e' is like e except that it crashes less often, then it satisfies all the contracts that e satisfies. This theorem is not used in proving the grand theorem, but it is an interesting property.

Theorem 7 (Monotonicity of Satisfaction) *For all e, e', t, if $e \in t$ and $e \preceq e'$, then $e' \in t$*

PROOF The proof begins by dealing with two special cases.

- Case $e \to^* \texttt{BAD}$: By definition of $\in$, we know $t = \texttt{Any}$. Since every expression satisfies `Any`, we have $e' \in \texttt{Any}$.
- Case $e\!\uparrow$: By Lemma 9^{p49} (b) (properties of $\preceq$), we know $e'\!\uparrow$. By definition of $\in$, we know $\forall t.\ e' \in t$.

Hence for the rest of the proof we assume that $e \to^* val \notin \{\texttt{BAD}, \texttt{UNR}\}$.

The rest of the proof is by induction on the size of t.

- Case $e \in \{x \mid p\}$: By Definition [A1], we know e is crash-free and $p[e/x] \not\to^* \{\texttt{BAD}, \texttt{False}\}$.

  By Lemma 9^{p49} (c) (properties of $\preceq$), e' is crash-free and has the same semantics as e. Thus, $e' \in t$.

- Case $e \in x\colon t_1 \to t_2$: We have the following induction hypothesis:

  $$\begin{array}{ll} \forall e_1 \in t_1.\ e_1 \preceq e_1' \Rightarrow e_1' \in t_1 & \text{[IH1]} \\ \forall e_2',\ \forall e_1 \in t_1, e_2 \in t_2[e_1/x].\ e_2 \preceq e_2' \Rightarrow e_2' \in t_2[e_1/x] & \text{[IH2]} \end{array}$$

  We also know that if $e \to^* \lambda x.e''$, then $e'\!\uparrow$ or $e' \to^* \lambda x.e''$.

  We have the following proof:

  $$\begin{array}{ll} & e \in t \text{ and } e \preceq e' \\ \Longleftrightarrow & \text{(By definition of } \in \text{ and by Lemma } 9^{\text{p49}} \text{ (d) (properties of } \preceq\text{))} \\ & e \to^* \lambda x.e'' \text{ and } \forall e_1 \in t_1.(e\ e_1) \in t_2[e_1/x] \text{ and} \\ & (e'\!\uparrow \text{ or } e' \to^* \lambda x.e'' \text{ and } e \preceq e') \end{array}$$

If $e'\uparrow$, by inspecting the definition of $\in$, we have $e' \in x\colon t_1 \to t_2$ as desired. Now we prove the case when e' does not diverge:

$$\begin{array}{rl} & e \to^* \lambda x.e_3 \text{ and } \forall e_1 \in t_1.(e\ e_1) \in t_2[e_1/x] \text{ and} \\ & e' \to^* \lambda x.e_4 \text{ and } e \preceq e' \\ \iff & \text{(By Theorem } 10^{\text{p93}} \text{ (congruence of } \preceq)) \\ & e \to^* \lambda x.e_3 \text{ and } \forall e_1 \in t_1.(e\ e_1) \in t_2[e_1/x] \text{ and} \\ & e' \to^* \lambda x.e_4 \text{ and } \forall e_5.\ (e\ e_5) \preceq (e'\ e_5) \\ \Rightarrow & \text{(By [IH2], choosing } e_5 \text{ to be } e_1 \in t_1) \\ & e' \to^* \lambda x.e_4 \text{ and } \forall e_1 \in t_1.\ (e'\ e_1) \in t_2[e_1/x] \\ \iff & \text{(By definition of } \in) \\ & e' \in x\colon t_1 \to t_2 \end{array}$$

Notice that, perhaps curiously, we do not use the induction hypothesis on t_1.

- Case $e \in (t_1, t_2)$: We have the following induction hypothesis

$$\begin{array}{ll} \forall e_1 \in t_1, e_1'.\ e_1 \preceq e_1' \Rightarrow e_1' \in t_1 & \text{[IH1]} \\ \forall e_2 \in t_2, e_2'.\ e_2 \preceq e_2' \Rightarrow e_2' \in t_2 & \text{[IH2]} \end{array}$$

  We also know that $e \to^* (e_1, e_2)$ and $e' \to^* (e_1, e_2)$.

  We have the following proof:

$$\begin{array}{rl} & (e_1, e_2) \in (t_1, t_2) \text{ and } e_1 \preceq e_1' \text{ and } e_2 \preceq e_2' \\ \iff & \text{(By definition of } \in) \\ & e_1 \in t_1 \text{ and } e_2 \in t_2 \text{ and } e_1 \preceq e_1' \text{ and } e_2 \preceq e_2' \\ \Rightarrow & \text{(By [IH1] and [IH2])} \\ & e_1' \in t_1 \text{ and } e_2' \in t_2 \\ \iff & \text{(By definition of } \in) \\ & (e_1', e_2') \in (t_1, t_2) \end{array}$$

- Case $e \in \texttt{Any}$: By definition of $\in$ i.e. [A4], any expression has type `Any`, so $e' \in \texttt{Any}$.∎

Chapter 6

Correctness Proofs of Contract Checking

To achieve the grand plan mentioned in Section 2.3 (i.e. to check $e \in t$), our static contract checking consists of three steps:

1. Construct the expression $e \triangleright t$ which captures all contract violations with `BAD`.
2. Simplify $e \triangleright t$ as much as possible, to e', say.
3. See if `BAD` is syntactically in e'; if not, e' is crash-free (i.e. no contract violations).

To justify the correctness of this approach, we need to prove Theorem 8[p81].

Theorem 8 (Soundness of Static Contract Checking) *For all closed expression e, and contract t,*

$$(\texttt{simpl}\ (e \triangleright t))\ \textit{is syntactically safe} \quad \Rightarrow \quad e \in t$$

Proof By Theorem 9[p81], Lemma 26[p107] and Lemma 3[p45] (e is syntactically safe $\Rightarrow$ e is crash-free).

The function `simpl` is defined by a set of semantically preserving simplification rules in the form of $e_1 \Longrightarrow e_2$, each of them satisfies Lemma 26[p107]. Details of symbolic simplification as well as the proof for Lemma 26[p107] are shown in Section 7.1.

Lemma 26[p107] (Correctness of One-Step Simplification) If $e_1 \Longrightarrow e_2$, then $e_1 \equiv_s e_2$.

The proof of Lemma 26[p107] can be found in Section 7.1.3. Theorem 9[p81] is exactly the same as Property 1[p63].

Theorem 9 (Soundness and Completeness of Contract Checking (grand theorem)) *For all closed expression e, closed and crash-free contract t,*

$$(e \triangleright t)\ \textit{is crash-free} \quad \Longleftrightarrow \quad e \in t$$

In this chapter, we give a detailed proof for Theorem 9[p81]. There are two directions to be proved:

- $e \in t \Rightarrow e \triangleright t$ is crash-free. The difficulty lies in the proof for dependent function contracts. We appeal to a key lemma (Lemma 19[p92] [Key Lemma] in Section 6.3).

- $e \triangleright t$ is crash-free $\Rightarrow e \in t$. The difficulty also lies in the proof for dependent function contracts. We appeal to three things:

  - definition and properties of crashes-more-often (Definition 7[p48], Lemma 9[p49]).
  - projection pair property of $\triangleright$ and $\triangleleft$ (Theorem 11[p94] in Section 6.6);
  - congruence of crashes-more-often (Theorem 10[p93] in Section 6.5).

We used to prove Lemma 19[p92] [Key Lemma] and Theorem 11[p94] [Projection Pair] by induction on the size of the contract t. But later on, we found that we could prove them with the help of a property named *telescoping property* in [BM06], though the telescoping property does not seem to be used in any of the proofs in [BM06]. Moreover, with the telescoping property (Section 6.2), we can prove the idempotency property of $\triangleright$ and $\triangleleft$ (Section 6.7) easily as well.

As mentioned in Section 5.2.1, the `fin` used in the proofs refers to the computable $\texttt{fin}_n$. That means if e diverges, then (`fin` e) converges to `True`.

As some of the proofs involve the structural induction on the size of contract, we define it in Figure 6.1.

$$
\begin{array}{lcl}
|.| & :: & \textbf{Contract} \to \textbf{Int} \\
|\{x \mid p\}| & = & 1 \\
|x\colon t_1 \to t_2| & = & |t_1| + |t_2| + 1 \\
|(t_1, t_2)| & = & |t_1| + |t_2| + 1 \\
|\texttt{Any}| & = & 1
\end{array}
$$

Figure 6.1: Size of Contract

To make the proof look less clustered, we use the following shorthands:

cf	:	crash-free
ss	:	syntatically safe
defn	:	definition
cl	:	closed

To make the dependency of theorems and lemmas clear, a dependency diagram is shown in Figure 6.2. For many theorems and lemmas, we prove them by induction on the size of contract t. The dashed directed edge shows that the size of the contract decreases, i.e. for a function contract $t_1 \to t_2$, we call another lemma (or theorem) with $t = t_1$. The solid directed edge shows the size of the contract is preserved. This makes the proof well-founded even though there are cycles in the dependencies (Section 6.4).

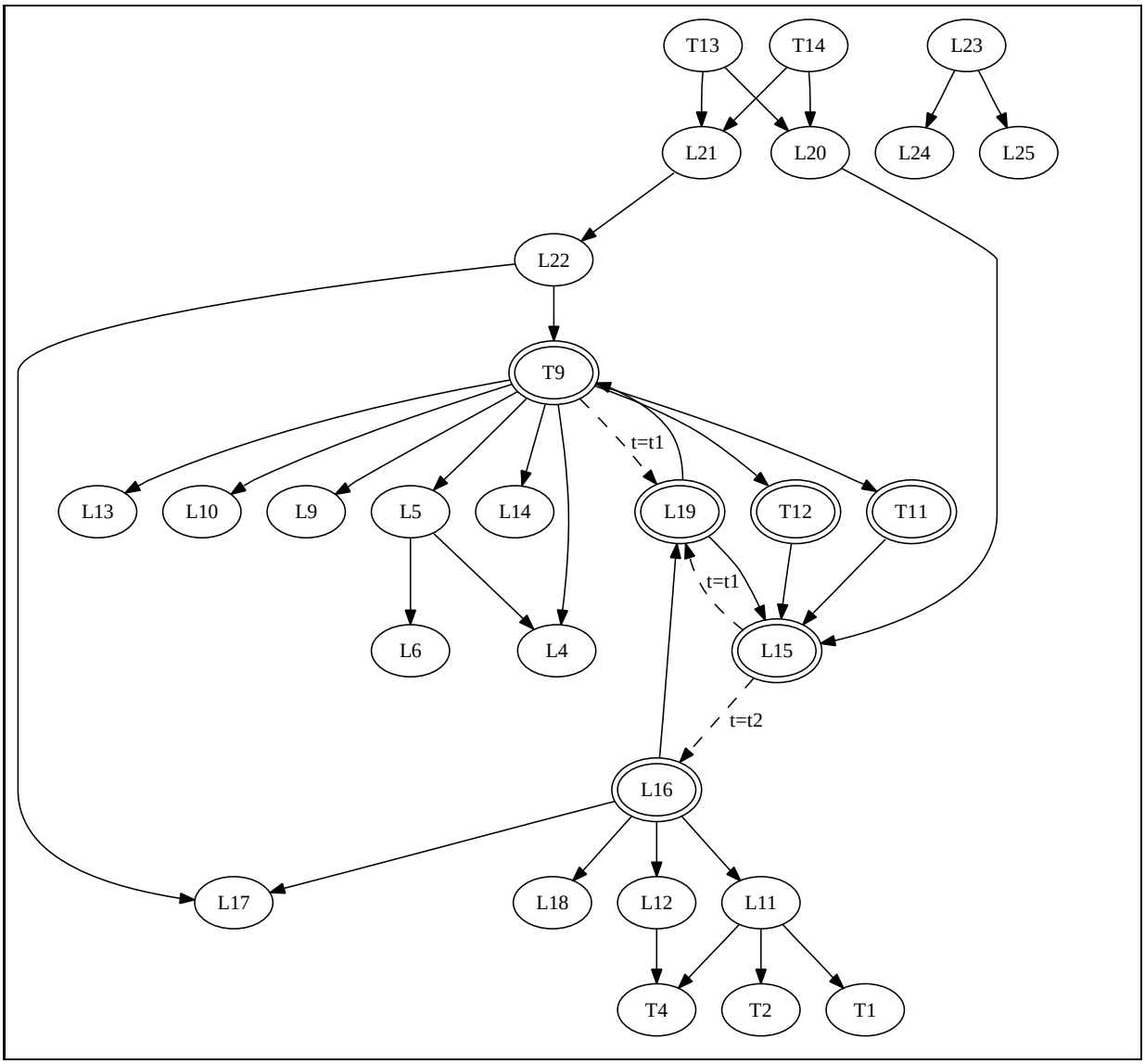

Figure 6.2: Dependency of Theorems and Lemmas

6.1 Proof of the Grand Theorem

Theorem $9^{\mathbf{p81}}$ For all expression e and crash-free contract t, such that $\vdash e :: \tau$ and $\vdash_c t :: \tau$,

$$e \triangleright t \text{ is crash-free} \iff e \in t$$

Proof The conditions $\vdash e :: \tau$ and $\vdash t :: \tau$ mean that both the expression e and the contract t are closed and well-typed. The proof begins by dealing with two special cases:

- Case $e \to^* \texttt{BAD}$: We prove the two directions separately.

($\Rightarrow$)

$e \triangleright t$ is **cf**
$\Rightarrow$ (By Lemma 4[p46] (preservation of crash-freeness) and Lemma 13[p66] (about `Any`))
$t = \texttt{Any}$
$\Rightarrow$ (By defn of $\in$, every expression satisfies `Any`)
$e \in t$

($\Leftarrow$)

$e \in t$
$\Rightarrow$ (By Lemma 4[p46] (preservation of crash-freeness) and Lemma 10[p55] (about `Any`))
$t = \texttt{Any}$
$\Rightarrow$ (By defn of $\triangleright$)
$e \triangleright \texttt{Any}$ is crash-free

- Case $e\uparrow$: By inspecting the definition of $\triangleright$ and $\in$, for all t, if $e\uparrow$, then $(e \triangleright t)\uparrow$ and $e \in t$. Thus, we are done.

Hence, for the rest of the proof, we assume that $e \to^* val \notin \{\texttt{BAD}, \texttt{UNR}\}$.

The rest of the proof is by induction on the size of t.

- Case t is $\{x \mid p\}$:

$e \triangleright \{x \mid p\}$ is **cf**
$\iff$ (By defn of $\triangleright$)
$\left(\begin{array}{l} e \text{ `}\texttt{seq}\text{`} \\ \quad \texttt{case fin } p[e/x] \texttt{ of} \\ \quad\quad \texttt{True} \to e \\ \quad\quad \texttt{False} \to \texttt{BAD} \end{array}\right)$ is **cf**
$\iff$ (Since $e \to^* val \notin \{\texttt{BAD}, \texttt{UNR}\}$, by defn of `` `seq` `` and `fin`)
e is **cf** and $\texttt{fin } p[e/x] \to^* \texttt{True}$
$\iff$ (By defn of `fin`)
e is **cf** and $p[e/x] \not\to^* \{\texttt{BAD}, \texttt{False}\}$
$\iff$ (By defn of $\in$)
$e \in \{x \mid p\}$

- Case t is $x\colon t_1 \to t_2$: we want to prove that

$$(e \triangleright x\colon t_1 \to t_2) \text{ is } \mathbf{cf} \iff e \in x\colon t_1 \to t_2$$

We have the following induction hypotheses:

$$\forall \mathbf{cl}\ e_1,\ e_1 \triangleright t_1 \text{ is } \mathbf{cf} \iff e_1 \in t_1 \quad \text{[IH1]}$$
$$\forall \mathbf{cl}\ e_2, e'.\ e_2 \triangleright t_2[e'/x] \text{ is } \mathbf{cf} \iff e_2 \in t_2[e'/x] \quad \text{[IH2]}$$

We have the following proof:

$e \triangleright x\colon t_1 \to t_2$ is **cf**.
$\iff$ (By defn of $\triangleright$)
e ‘`seq`‘ $\lambda v.(e\ (v \triangleleft t_1)) \triangleright t_2[(v \triangleleft t_1)/x]$ is **cf**.
$\iff$ (Since $e \to^* val \notin \{\texttt{BAD}, \texttt{UNR}\}$, by Lemma 14[p67](c) (properties of `seq`))
$\lambda v.\ (e\ (v \triangleleft t_1)) \triangleright t_2[(v \triangleleft t_1)/x]$ is **cf**.
$\iff$ (By Lemma 5[p46] (crash-free function))
(†) $\forall \mathbf{cf}\ e'.\ (e\ (e' \triangleleft t_1)) \triangleright t_2[(e' \triangleleft t_1)/x]$ is **cf**.

Now the proof splits into two. In the reverse direction, we start with the assumption $e \in x\colon t_1 \to t_2$:

$e \in x\colon t_1 \to t_2$
$\iff$ (By defn of $\in$)
$\forall\, e_1 \in t_1.\ (e\ e_1) \in t_2[e_1/x]$
$\Rightarrow$ (By Lemma 19[p92] (Key Lemma), let $e_1 = e' \triangleleft t_1$)
$\forall \mathbf{cf}\ e'.\ (e\ (e' \triangleleft t_1)) \in t_2[(e' \triangleleft t_1)/x])$
$\iff$ (By [IH2])
(†) $\forall \mathbf{cf}\ e'.\ (e\ (e' \triangleleft t_1)) \triangleright t_2[(e' \triangleleft t_1)/x]$ is **cf**.

Now we have reached the desired conclusion (†). The key step is the use of Lemma 19[p92] (the first key lemma) (see §6.3).

In the forward direction, we start with (†):

$\forall \mathbf{cf}\ e'.\ (e\ (e' \triangleleft t_1)) \triangleright t_2[(e' \triangleleft t_1)/x]$ is **cf**.
$\Rightarrow$ (By [IH1], $e_1 \in t_1 \Rightarrow (e_1 \triangleright t_1)$ is **cf**
so we replace e' by $e_1 \triangleright t_1$)
$\forall e_1 \in t_1.\ (e\ ((e_1 \triangleright t_1) \triangleleft t_1)) \triangleright t_2[(e_1 \triangleright t_1 \triangleleft t_1)/x]$ is **cf**
$\Rightarrow$ (By (Theorem 11[p94] (projection pair) and
Theorem 10[p93] (congruence of $\preceq$) and
Lemma 9[p49] (c) (property of $\preceq$)) twice)
$\forall e_1 \in t_1.\ (e\ e_1) \triangleright t_2[e_1/x])$ is **cf**
$\Rightarrow$ (By [IH2])
$\forall\, e_1 \in t_1.\ (e\ e_1) \in t_2[e_1/x])$
$\iff$ (by definition of $\in$)
$e \in x\colon t_1 \to t_2$

There are two key steps: one is to choose a *particular* crash-free e', namely $(e_1 \triangleright t_1)$ where $e_1 \in t_1$; the other one is the appeal to Theorem 11[p94], the projection pair property of $\triangleright$ and $\triangleleft$.

- t is (t_1, t_2): We have the following induction hypotheses:

$\forall \mathbf{cl}\ e_1, t_1.\ e_1 \triangleright t_1$ is **cf** $\iff e_1 \in t_1$ [IH1]
$\forall \mathbf{cl}\ e_2, t_2.\ e_2 \triangleright t_2$ is **cf** $\iff e_2 \in t_2$ [IH2]

We have

$$
\begin{array}{ll}
 & e \triangleright (t_1, t_2) \text{ is } \mathbf{cf} \\
\iff & \text{(By defn of } \triangleright\text{)} \\
 & \texttt{case } e \texttt{ of } \{(e_1, e_2) \to (e_1 \triangleright t_1, e_2 \triangleright t_2)\} \text{ is } \mathbf{cf} \\
\iff & \text{(By [E-match1] and defn of } \mathbf{cf}\text{)} \\
 & e \to^* (e_1, e_2) \text{ and} \\
 & (e_1 \triangleright t_1) \text{ is } \mathbf{cf} \text{ and } (e_2 \triangleright t_2) \text{ is } \mathbf{cf} \\
\iff & \text{(By [IH1] and [IH2])} \\
 & e \to^* (e_1, e_2) \text{ and } e_1 \in t_1 \text{ and } e_2 \in t_2 \\
\iff & \text{(By definition of } \in\text{)} \\
 & e \in (t_1, t_2)
\end{array}
$$

- t is `Any`: We have:

$$
\begin{array}{ll}
 & e \triangleright \texttt{Any} \text{ is } \mathbf{cf} \\
\iff & \text{(By definition of } \triangleright\text{)} \\
 & \texttt{UNR} \text{ is } \mathbf{cf} \\
\iff & \text{(By definition of } \in, \text{ and } \texttt{UNR} \in \texttt{Any}\text{)} \\
 & e \in \texttt{Any}
\end{array}
$$

End of proof. ∎

6.2 Telescoping Property

The telescoping property is adopted from [BM06] and we found that this property makes the proofs of many lemmas shorter. However, it is not used in any proof in [BM06].

Lemma 15 (Telescoping Property) *For all e, and crash-free t,* $(e \underset{r_2}{\overset{r_1}{\bowtie}} t) \underset{r_4}{\overset{r_3}{\bowtie}} t = e \underset{r_4}{\overset{r_1}{\bowtie}} t$

Proof Before we start the proof, by defn of ‘`seq`‘and [E-case1], we know two facts:

[Fact1] $\forall e'.$ `BAD` ‘`seq`‘ $e' \to^*$ `BAD`

[Fact2] $\forall alts,$ `case BAD of` $alts \to$ `BAD`

The proof begins by dealing with two special cases.

- Case $e \to^*$ `BAD`: Based on [Fact1] and [Fact2], for all $t \neq$ `Any`, by inspecting the definition of $\bowtie$, we know $(e \underset{r_j}{\overset{r_i}{\bowtie}} t) \to^*$ `BAD`$\forall i, j$. So LHS=RHS=`BAD` for $t \neq$ `Any`. In the case $t =$ `Any`, we have:

$$
\begin{array}{ll}
 & (e \underset{r_2}{\overset{r_1}{\bowtie}} \texttt{Any}) \underset{r_4}{\overset{r_3}{\bowtie}} \texttt{Any} \\
= & r_2 \underset{r_4}{\overset{r_3}{\bowtie}} \texttt{Any} \\
= & r_4 \\
= & e \underset{r_4}{\overset{r_3}{\bowtie}} \texttt{Any}
\end{array}
$$

- $e\uparrow$. Similar to the arguments in the case $e \to^* \texttt{BAD}$.

Hence for the rest of the proof we assume that $e \to^* val \notin \{\texttt{BAD}, \texttt{UNR}\}$.
The rest of the proof is by induction on the size of t.

- t is $\{x \mid p\}$:

$$
\begin{array}{ll}
 & (e \overset{r_1}{\underset{r_2}{\bowtie}} \{x \mid p\}) \overset{r_3}{\underset{r_4}{\bowtie}} \{x \mid p\} \\
= & \text{(By definition of } \bowtie) \\
 & \texttt{let } y = \texttt{let } x = e \texttt{ in } \left(\begin{array}{l} x \text{ `}\texttt{seq}\text{`} \texttt{ case fin } p \texttt{ of} \\ \qquad \texttt{True} \to x \\ \qquad \texttt{False} \to r_1 \end{array} \right) \\
 & \texttt{in } y \text{ `}\texttt{seq}\text{`} \texttt{ case fin } p[y/x] \texttt{ of} \\
 & \qquad \texttt{True} \to y \\
 & \qquad \texttt{False} \to r_3 \\
= & \text{(Since } y \text{ is strict due to `}\texttt{seq}\text{`, we float } \texttt{let } x = e \text{ out)} \\
 & \texttt{let } x = e \\
 & \texttt{in } x \text{ `}\texttt{seq}\text{`} \texttt{ case fin } p \texttt{ of} \\
 & \qquad \texttt{True} \to \texttt{ let } y = x \\
 & \qquad\qquad \texttt{in } y \text{ `}\texttt{seq}\text{`} \texttt{ case fin } p[y/x] \texttt{ of} \\
 & \qquad\qquad\qquad \texttt{True} \to y \\
 & \qquad\qquad\qquad \texttt{False} \to r_3 \\
 & \qquad \texttt{False} \to \texttt{ let } y = r_1 \\
 & \qquad\qquad \texttt{in } y \text{ `}\texttt{seq}\text{`} \texttt{ case fin } p[y/x] \texttt{ of} \\
 & \qquad\qquad\qquad \texttt{True} \to y \\
 & \qquad\qquad\qquad \texttt{False} \to r_3 \\
= & \text{(inline } x \text{ and inline } y) \\
 & e \text{ `}\texttt{seq}\text{`} \texttt{ case fin } p[e/x] \texttt{ of} \\
 & \qquad \texttt{True} \to \texttt{ } e \text{ `}\texttt{seq}\text{`} \texttt{ case fin } p[e/x] \texttt{ of} \\
 & \qquad\qquad\qquad \texttt{True} \to e \\
 & \qquad\qquad\qquad \texttt{False} \to r_3 \\
 & \qquad \texttt{False} \to \texttt{ } r_1 \text{ `}\texttt{seq}\text{`} \texttt{ case fin } p[e/x] \texttt{ of} \\
 & \qquad\qquad\qquad \texttt{True} \to r_1 \\
 & \qquad\qquad\qquad \texttt{False} \to r_3 \\
= & \text{(Since } e \to^* val \notin \{\texttt{BAD}, \texttt{UNR}\} \text{ and } r_1 \in \{\texttt{BAD}, \texttt{UNR}\} \\
 & \text{and propagating the } \texttt{True} \text{ (or } \texttt{False}\text{) value of } \texttt{fin } p[e/x] \text{to sub-branches)} \\
 & e \text{ `}\texttt{seq}\text{`} \texttt{ case fin } p[e/x] \texttt{ of} \\
 & \qquad \texttt{True} \to e \\
 & \qquad \texttt{False} \to r_1 \\
= & \text{(By defn of } \bowtie) \\
 & e \overset{r_1}{\underset{r_4}{\bowtie}} t
\end{array}
$$

- t is $x\colon t_1 \to t_2$: We have the following induction hypotheses:

$$\forall e, \mathbf{cf}\ t_1,\ (e \overset{r_1}{\underset{r_2}{\bowtie}} t_1) \overset{r_3}{\underset{r_4}{\bowtie}} t_1 = e \overset{r_1}{\underset{r_4}{\bowtie}} t_1 \qquad \text{[IH1]}$$

$$\forall e, e', \mathbf{cf}\ t_2[e'/x],\ (e \overset{r_1}{\underset{r_2}{\bowtie}} t_2[e'/x]) \overset{r_3}{\underset{r_4}{\bowtie}} t_2[e'/x] = e \overset{r_1}{\underset{r_4}{\bowtie}} t_2[e'/x] \qquad \text{[IH2]}$$

We have the following proof:

$$
\begin{array}{ll}
 & (e \overset{r_1}{\underset{r_2}{\bowtie}} x\colon t_1 \to t_2) \overset{r_3}{\underset{r_4}{\bowtie}} x\colon t_1 \to t_2 \\
= & \text{(By defn of } \bowtie\text{)} \\
 & \lambda v_1.\ ((e \overset{r_1}{\underset{r_2}{\bowtie}} x\colon t_1 \to t_2) \quad (v_1 \overset{r_4}{\underset{r_3}{\bowtie}} t_1)) \overset{r_3}{\underset{r_4}{\bowtie}} t_2[(v_1 \overset{r_4}{\underset{r_3}{\bowtie}} t_1)/x] \\
= & \text{(By defn of } \bowtie \text{ again)} \\
 & \lambda v_1.\ ((\lambda v_2.\ (e\ (v_2 \overset{r_2}{\underset{r_1}{\bowtie}} t_1)) \overset{r_1}{\underset{r_2}{\bowtie}} t_2[(v_2 \overset{r_2}{\underset{r_1}{\bowtie}} t_1)/x]) \quad (v_1 \overset{r_4}{\underset{r_3}{\bowtie}} t_1)) \overset{r_3}{\underset{r_4}{\bowtie}} t_2[(v_1 \overset{r_4}{\underset{r_3}{\bowtie}} t_1)/x] \\
= & \text{(By } \beta\text{-reduction)} \\
 & \lambda v_1.\ ((e\ ((v_1 \overset{r_4}{\underset{r_3}{\bowtie}} t_1) \overset{r_2}{\underset{r_1}{\bowtie}} t_1)) \overset{r_1}{\underset{r_2}{\bowtie}} t_2[(v_1 \overset{r_4}{\underset{r_3}{\bowtie}} t_1 \overset{r_2}{\underset{r_1}{\bowtie}} t_1)/x]) \overset{r_3}{\underset{r_4}{\bowtie}} t_2[(v_1 \overset{r_4}{\underset{r_3}{\bowtie}} t_1)/x] \\
= & \text{(By induction hypothesis with } t = t_1\text{)} \\
 & \lambda v_1.\ ((e\ (v_1 \overset{r_4}{\underset{r_1}{\bowtie}} t_1)) \overset{r_1}{\underset{r_2}{\bowtie}} t_2[(v_1 \overset{r_4}{\underset{r_1}{\bowtie}} t_1)/x]) \overset{r_3}{\underset{r_4}{\bowtie}} t_2[(v_1 \overset{r_4}{\underset{r_3}{\bowtie}} t_1)/x] \\
= & \text{(By Lemma 16}^{\text{p89}}\text{, we replace } r_3 \text{ by } r_1\text{)} \\
 & \lambda v_1.\ ((e\ (v_1 \overset{r_4}{\underset{r_1}{\bowtie}} t_1)) \overset{r_1}{\underset{r_2}{\bowtie}} t_2[(v_1 \overset{r_4}{\underset{r_1}{\bowtie}} t_1)/x]) \overset{r_3}{\underset{r_4}{\bowtie}} t_2[(v_1 \overset{r_4}{\underset{r_1}{\bowtie}} t_1)/x] \\
= & \text{By induction hypothesis [IH2]: } t = t_2[(v_1 \overset{r_4}{\underset{r_1}{\bowtie}} t_1)/x] \\
 & t_2[(v_1 \overset{r_4}{\underset{r_1}{\bowtie}} t_1)/x] \text{is } \mathbf{cf} \text{ because } t \text{ is } \mathbf{cf} \text{ and by Lemma 19}^{\text{p92}}\text{, } v_1 \overset{r_4}{\underset{r_1}{\bowtie}} t_1 \in t_1\text{)} \\
 & \lambda v_1.\ (e\ (v_1 \overset{r_4}{\underset{r_1}{\bowtie}} t_1)) \overset{r_1}{\underset{r_4}{\bowtie}} t_2[(v_1 \overset{r_4}{\underset{r_1}{\bowtie}} t_1)/x] \\
= & \text{(By defn of } \bowtie\text{)} \\
 & e \overset{r_1}{\underset{r_4}{\bowtie}} x\colon t_1 \to t_2
\end{array}
$$

- t is (t_1, t_2): We have the following induction hypotheses:

$$
\begin{array}{l}
\forall e, \mathbf{cf}\ t_1,\ (e \overset{r_1}{\underset{r_2}{\bowtie}} t_1) \overset{r_3}{\underset{r_4}{\bowtie}} t_1 = e \overset{r_1}{\underset{r_4}{\bowtie}} t_1 \quad \text{[IH1]} \\
\forall e, \mathbf{cf}\ t_2,\ (e \overset{r_1}{\underset{r_2}{\bowtie}} t_2) \overset{r_3}{\underset{r_4}{\bowtie}} t_2 = e \overset{r_1}{\underset{r_4}{\bowtie}} t_2 \quad \text{[IH2]}
\end{array}
$$

We have the following proof:

$$
\begin{array}{ll}
 & (e \overset{r_1}{\underset{r_2}{\bowtie}} (t_1, t_2)) \overset{r_3}{\underset{r_4}{\bowtie}} (t_1, t_2) \\
= & \text{(By defn of projection)} \\
 & \left(\begin{array}{l} \texttt{case}\ e\ \texttt{of} \\ (e_1, e_2) \to (e \overset{r_1}{\underset{r_2}{\bowtie}} t_1, e \overset{r_1}{\underset{r_2}{\bowtie}} t_2) \end{array} \right) \overset{r_3}{\underset{r_4}{\bowtie}} (t_1, t_2) \\
= & \text{(By simpl rule CaseOut)} \\
 & \begin{array}{l} \texttt{case}\ e\ \texttt{of} \\ (e_1, e_2) \to ((e \overset{r_1}{\underset{r_2}{\bowtie}} t_1) \overset{r_3}{\underset{r_4}{\bowtie}} t_1, (e \overset{r_1}{\underset{r_2}{\bowtie}} t_2) \overset{r_3}{\underset{r_4}{\bowtie}} t_2) \end{array} \\
= & \text{(By induction hypotheses [IH1] and [IH2])} \\
 & \begin{array}{l} \texttt{case}\ e\ \texttt{of} \\ (e_1, e_2) \to (e \overset{r_1}{\underset{r_4}{\bowtie}} t_1, e \overset{r_1}{\underset{r_4}{\bowtie}} t_2) \end{array} \\
= & \text{(By defn of projection)} \\
 & e \overset{r_1}{\underset{r_4}{\bowtie}} (t_1, t_2)
\end{array}
$$

- t is `Any`:

  LHS

$$\begin{aligned} & (e \overset{r_1}{\underset{r_2}{\bowtie}} \texttt{Any}) \overset{r_3}{\underset{r_4}{\bowtie}} \texttt{Any} \\ = \; & r_2 \overset{r_3}{\underset{r_4}{\bowtie}} \texttt{Any} \\ = \; & r_4 \end{aligned}$$

  RHS

$$\begin{aligned} & e \overset{r_3}{\underset{r_4}{\bowtie}} \texttt{Any} \\ = \; & r_4 \end{aligned}$$

  Since LHS $\equiv$ RHS, we are done. ∎

In the case of dependent function contract $x\colon t_1 \to t_2$, in order to apply the induction hypothesis for t_2, we need a crucial lemma (Lemma 16[p89]) which says that the lower exception in t_2 is not reachable, so we can replace r_3 by r_1. The intuition is that since $x\colon t_1 \to t_2$ is a crash-free contract, t_2 must use x in a way that respects the contract t_1, so wrapping x in a contract that raises exception r_1 if the context does not respect t_1 cannot cause r_1 to be raised.

Lemma 16 (Unreachable Exception) *For all t_1, t_2, if $\forall e_1 \in t_1$, $t_2[e_1/x]$ is crash-free, then for all r_1, r_3, r_4, for all crash-free e_1', $t_2[(e_1' \overset{r_4}{\underset{r_3}{\bowtie}} t_1)/x] \equiv_t t_2[(e_1' \overset{r_4}{\underset{r_1}{\bowtie}} t_1)/x]$*

Proof We prove this by induction on the size of t_2. Note that, by the definition of the size of a contract and the definition of contract substitution, the contract substitution preserves the size of the contract.

- Case $t_2 = \{y \mid p\}$. We have the following proof:

$$
\begin{array}{ll}
 & \forall e_1 \in t_1.\ \{y \mid p\}[e_1/x] \text{ is } \mathbf{cf} \\
\iff & \text{(By substitution)} \\
 & \forall e_1 \in t_1.\ \{y \mid p[e_1/x]\} \text{ is } \mathbf{cf} \\
\iff & \text{(By defn of } \mathbf{cf}\text{)} \\
 & \forall e_1 \in t_1.\ p[e_1/x] \text{ is } \mathbf{cf} \\
\Rightarrow & \text{(By Lemma 19}^{\text{p92}}\text{ (Key Lemma), } \forall \mathbf{cf}\ e_1'.\ e_1' \triangleleft t_1 \in t_1) \\
 & \forall \mathbf{cf} e_1'.\ p[(e_1' \triangleleft t_1)/x] \text{ is } \mathbf{cf} \\
\iff & \text{(By defn of } \triangleleft\text{)} \\
 & \forall \mathbf{cf} e_1'.\ p[(e_1' \overset{\text{UNR}}{\underset{\text{BAD}}{\bowtie}} t_1)/x] \text{ is } \mathbf{cf} \\
\iff & \text{(By defn of } \mathbf{cf} \text{ and } p[(e_1' \overset{\text{UNR}}{\underset{\text{BAD}}{\bowtie}} t_1)/x] \text{ has type } \texttt{Bool}\text{)} \\
 & \forall \mathbf{cf} e_1'.\ p[(e_1' \overset{\text{UNR}}{\underset{\text{BAD}}{\bowtie}} t_1)/x] \not\to^* \texttt{BAD} \\
\iff & \text{(Let } \mathcal{D} = p[(e_1' \overset{\bullet_1}{\underset{\bullet_2}{\bowtie}} t_1)/x]\text{)} \\
 & \mathcal{D}[\![\texttt{UNR}, \texttt{BAD}]\!] \not\to^* \texttt{BAD} \\
\Rightarrow & \text{(By Lemma 17}^{\text{p91}}\text{ (Exception I))} \\
 & \forall r_1, r_3,\ \mathcal{D}[\![\texttt{UNR}, r_1]\!] \equiv_s \mathcal{D}[\![\texttt{UNR}, r_3]\!] \\
\Rightarrow & \text{(By Lemma 18}^{\text{p91}}\text{ (Exception II))} \\
 & \forall r_1, r_3, r_4,\ \mathcal{D}[\![r_4, r_1]\!] \equiv_s \mathcal{D}[\![r_4, r_3]\!] \\
\iff & \text{(By definition of } \mathcal{D}\text{)} \\
 & \forall \mathbf{cf}\ e_1', r_1, r_3, r_4.\ p[(e_1' \overset{r_4}{\underset{r_3}{\bowtie}} t_1)/x] \equiv_s p[(e_1' \overset{r_4}{\underset{r_1}{\bowtie}} t_1)/x] \\
\iff & \text{(By Lemma 11}^{\text{p61}}\text{ (Predicate Contract Equivalence) and substitution)} \\
 & \forall \mathbf{cf}\ e_1', r_1, r_3, r_4.\ \{y \mid p\}[(e_1' \overset{r_4}{\underset{r_3}{\bowtie}} t_1)/x] \equiv_t \{y \mid p\}[(e_1' \overset{r_4}{\underset{r_1}{\bowtie}} t_1)/x]
\end{array}
$$

- Case $t_2 = y\colon t_3 \to t_4$. We have the following induction hypotheses:

$$
\begin{array}{ll}
 & \forall e_1 \in t_1,\ (t_3[e_1/x] \text{ is } \mathbf{cf} \Rightarrow \forall \mathbf{cf}\ e_1',\ t_3[(e_1' \overset{r_4}{\underset{r_3}{\bowtie}} t_1)/x] \equiv_t t_3[(e_1' \overset{r_4}{\underset{r_1}{\bowtie}} t_1)/x]) \quad \text{[IH1]} \\
 & \forall e_1 \in t_1, \forall e_3 \in t_3,\ (t_4[e_3/y, e_1/x] \text{ is } \mathbf{cf} \Rightarrow \\
 & \forall \mathbf{cf}\ e_1',\ t_4[e_3/y, (e_1' \overset{r_4}{\underset{r_3}{\bowtie}} t_1)/x] \equiv_t t_4[e_3/y, (e_1' \overset{r_4}{\underset{r_1}{\bowtie}} t_1)/x]) \quad \text{[IH2]}
\end{array}
$$

  We have the following proof:

$$
\begin{array}{ll}
 & \forall e_1 \in t_1,\ (y\colon t_3 \to t_4)[e_1/x] \text{ is } \mathbf{cf} \\
\iff & \text{(By defn of } \mathbf{cf}\text{)} \\
 & \forall e_1 \in t_1.\ t_3[e_1/x] \text{ is } \mathbf{cf} \text{ and } \forall e_3 \in t_3[e_1/x],\ t_4[e_1/x, e_3/y] \text{ is } \mathbf{cf} \\
\iff & \text{(By [IH1] and [IH2])} \\
 & \forall \mathbf{cf} e_1',\ t_3[(e_1' \overset{r_4}{\underset{r_3}{\bowtie}} t_1)/x] \equiv_t t_3[(e_1' \overset{r_4}{\underset{r_1}{\bowtie}} t_1)/x] \text{ and} \\
 & \forall e_3 \in t_3[e_1/x], \forall \mathbf{cf} e_1',\ t_4[e_3/y, (e_1' \overset{r_4}{\underset{r_3}{\bowtie}} t_1)/x] \equiv_t t_4[e_3/y, (e_1' \overset{r_4}{\underset{r_1}{\bowtie}} t_1)/x] \\
\iff & \text{(By Lemma 12}^{\text{p61}}\text{ (Dependent Function Contract Equivalence) and} \\
 & \text{contract substitution in Figure 4.5)} \\
 & \forall \mathbf{cf} e_1',\ y\colon t_3 \to t_4[(e_1' \overset{r_4}{\underset{r_3}{\bowtie}} t_1)/x] \equiv_t y\colon t_3 \to t_4[(e_1' \overset{r_4}{\underset{r_1}{\bowtie}} t_1)/x]
\end{array}
$$

- Case $t_2 = (t_3, t_4)$. We have the following induction hypotheses:

$$\forall e_1 \in t_1,\ t_3[e_1/x] \text{ is } \mathbf{cf} \ \Rightarrow\ \forall \mathbf{cf}\ e_1',\ t_3[(e_1' \underset{r_3}{\overset{r_4}{\bowtie}} t_1)/x] \equiv_t t_3[(e_1' \underset{r_1}{\overset{r_4}{\bowtie}} t_1)/x] \quad \text{[IH1]}$$
$$\forall e_1 \in t_1,\ t_4[e_1/x] \text{ is } \mathbf{cf} \ \Rightarrow\ \forall \mathbf{cf}\ e_1',\ t_4[(e_1' \underset{r_3}{\overset{r_4}{\bowtie}} t_1)/x] \equiv_t t_4[(e_1' \underset{r_1}{\overset{r_4}{\bowtie}} t_1)/x] \quad \text{[IH2]}$$

  We have the following proof:

$$\begin{array}{ll} & \forall e_1 \in t_1.\ (t_3, t_4)[e_1/x] \text{ is } \mathbf{cf}) \\ \iff & \text{(By substitution)} \\ & \forall e_1 \in t_1.\ (t_3[e_1/x], t_4[e_1/x]) \text{ is } \mathbf{cf}) \\ \iff & \text{(By defn of } \mathbf{cf}) \\ & \forall e_1 \in t_1.\ t_3[e_1/x] \text{ is } \mathbf{cf} \text{ and } t_4[e_1/x]) \text{ is } \mathbf{cf}) \\ \iff & \text{(By [IH1] and [IH2])} \\ & \forall \mathbf{cf}\ e_1',\ t_3[(e_1' \underset{r_3}{\overset{r_4}{\bowtie}} t_1)/x] \equiv_t t_3[(e_1' \underset{r_1}{\overset{r_4}{\bowtie}} t_1)/x] \text{ and } t_4[(e_1' \underset{r_3}{\overset{r_4}{\bowtie}} t_1)/x] \equiv_t t_4[(e_1' \underset{r_1}{\overset{r_4}{\bowtie}} t_1)/x] \\ \iff & \text{(By substitution)} \\ & \forall \mathbf{cf}\ e_1',\ (t_3, t_4)[(e_1' \underset{r_3}{\overset{r_4}{\bowtie}} t_1)/x] \equiv_t (t_3, t_4)[(e_1' \underset{r_1}{\overset{r_4}{\bowtie}} t_1)/x] \end{array}$$

- Case $t_2 = \texttt{Any}$. Immediate result. ∎

In Lemma 17[p91], we prove the equivalence of two terms under a more general condition $\mathcal{C}[\![\texttt{BAD}]\!]$ is **cf** rather than simply $\mathcal{C}[\![\texttt{BAD}]\!] \not\rightarrow^* \texttt{BAD}$ (which is good enough for proving Lemma 16[p89] (Unreachable Exception)) because Lemma 17[p91] is also used in proving Lemma 22[p96] (Exception III).

Lemma 17 (Exception I) $\forall \mathcal{C}.\ (\mathcal{C}[\![\texttt{BAD}]\!]\ \textit{is}\ \mathbf{cf} \Rightarrow \forall r \in \{\texttt{BAD}, \texttt{UNR}\}.\ \mathcal{C}[\![r]\!] \equiv_s \mathcal{C}[\![\texttt{BAD}]\!])$

Proof There are only two exceptional values: `BAD` and `UNR`. Hence, it is to prove that:

$$\forall \mathcal{C}, \quad (\mathcal{C}[\![\texttt{BAD}]\!] \text{ is } \mathbf{cf} \quad \Rightarrow \quad \mathcal{C}[\![\texttt{UNR}]\!] \equiv_s \mathcal{C}[\![\texttt{BAD}]\!])$$

The intuition is that if $\mathcal{C}[\![\texttt{BAD}]\!]$ is **cf**, then the `BAD` in the hole cannot be reached, so we can replace it by any exceptional value. This reasoning in turn relies on the absence of a "catch" primitive that can transform `BAD` into something non-`BAD`. Note that the following may hold:

$$\forall \mathcal{C}, e_1, e_2.\ (\mathcal{C}[\![\texttt{BAD}]\!] \text{ is } \mathbf{cf} \Rightarrow \mathcal{C}[\![e_1]\!] \equiv_s \mathcal{C}[\![e_2]\!])$$

but in this thesis, we only need to prove this specific lemma.

Formally, we can prove the lemma by case splitting on whether $\mathcal{C}[\![\texttt{BAD}]\!]$ terminates, and if it does, by induction on the number of steps of reduction. ∎

Lemma 18 (Exception II) $\forall \mathcal{C}, r_1, r_3, r_4.\ \mathcal{C}[\![\texttt{UNR}, r_1]\!] \equiv_s \mathcal{C}[\![\texttt{UNR}, r_3]\!] \Rightarrow \mathcal{C}[\![r_4, r_1]\!] \equiv_s \mathcal{C}[\![r_4, r_3]\!]$

Proof There are only two exceptional values: `BAD` and `UNR`. Hence, it is to prove that:

$$\forall \mathcal{C}, r_1, r_3.\ \mathcal{C}[\![\texttt{UNR}, r_1]\!] \equiv_s \mathcal{C}[\![\texttt{UNR}, r_3]\!] \Rightarrow \mathcal{C}[\![\texttt{BAD}, r_1]\!] \equiv_s \mathcal{C}[\![\texttt{BAD}, r_3]\!]$$

We can see that there are two holes and the first hole is filled in with the same value while the second hole is filled in with different exceptional values indicated by r_1 and r_3 respectively. It is obvious that $\forall \mathcal{C}.\ \mathcal{C}[\![\texttt{BAD}]\!] \equiv_s \mathcal{C}[\![\texttt{BAD}]\!]$ because two sides are syntactically the same. The intuition is that even if we have two holes (instead of one hole), as long as the second hole does not play a role in any reduction i.e. unreachable, the equality $\equiv_s$ still holds. We use the constraint $\mathcal{C}[\![\texttt{UNR}, r_1]\!] \equiv_s \mathcal{C}[\![\texttt{UNR}, r_3]\!]$ to specify the condition that the second hole does not play a role.

Formally, we can prove the lemma by case splitting on whether $\mathcal{C}[\![r_4, r_1]\!]$ terminates, and if it does, by induction on the number of steps of reduction.

6.3 Key Lemma

Lemma 19 (Key Lemma) *For all crash-free e and crash-free contract t, such that $\vdash e :: \tau$ and $\vdash_c t :: \tau$,*

$$e \lhd t \ \in\ t$$

Proof First, we have the following derivation (named D1).

$$\begin{array}{rl} & (e \lhd t) \rhd t \\ = & \text{(By defn of } \lhd \text{ and } \rhd\text{)} \\ & (e \underset{\texttt{BAD}}{\overset{\texttt{UNR}}{\bowtie}} t) \underset{\texttt{UNR}}{\overset{\texttt{BAD}}{\bowtie}} t \\ = & \text{(By Lemma 15}^{\text{p86}}\text{ (Telescoping Property))} \\ & e \underset{\texttt{UNR}}{\overset{\texttt{UNR}}{\bowtie}} t \end{array}$$

Now, we have the following proof.

$$\begin{array}{rl} & e \text{ is } \mathbf{cf} \\ \Rightarrow & \text{(Since } t \text{ is crash-free, } t \equiv \lfloor t \rfloor\text{. By the defn of } \bowtie\text{,} \\ & \text{the context } (\bullet \underset{\texttt{UNR}}{\overset{\texttt{UNR}}{\bowtie}} \lfloor t \rfloor) \text{ is syntactically safe.} \\ & \text{By defn of } \mathbf{cf}\text{, we have below)} \\ & e \underset{\texttt{UNR}}{\overset{\texttt{UNR}}{\bowtie}} t \text{ is } \mathbf{cf} \\ \Longleftrightarrow & \text{(By derivation D1)} \\ & (e \lhd t) \rhd t \text{ is } \mathbf{cf} \\ \Longleftrightarrow & \text{(By Theorem 9}^{\text{p81}}\text{ (grand theorem))} \\ & (e \lhd t) \in t \end{array}$$

End of proof. ∎

6.4 Examination of Cyclic Dependencies

Recall the dependency graph in Figure 6.2, there are a few cycles:

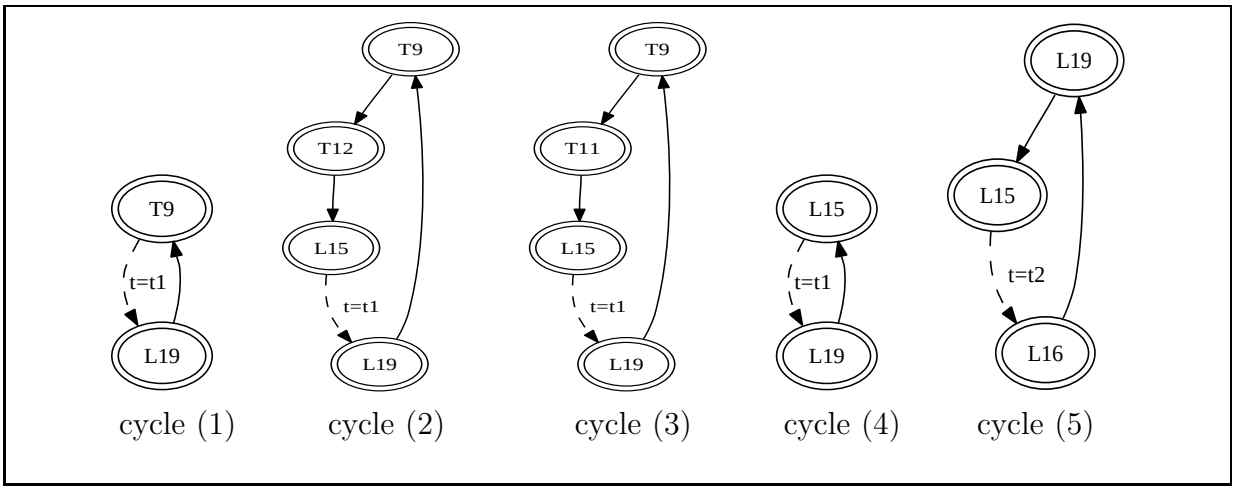

Figure 6.3: Cyclic Dependency of Three Lemmas

(1) T5 → L19 → T5

(2) T5 → T12 → L15 → L19 → T5

(3) T5 → T11 → L15 → L19 → T5

(4) L19 → L15 → L19

(5) L19 → L15 → L16 → L19

Each cycle is shown in Figure 6.3. The dashed directed edge indicates a decrease in size of t while the solid directed edge shows a preservation of the size of t. We can see that, in each cycle, there is an edge that decreases the size of t. Cycle (1) is well-founded because the size of t (where $t = x\colon t_1 \to t_2$) decreases (to t_1) when Theorem 9^{p81} calls Lemma 19^{p92}. Cycle (2), Cycle (3) and Cycle (4) are well-founded because the size of t (where $t = x\colon t_1 \to t_2$) decreases (to t_1) when Lemma 15^{p86} calls Lemma 19^{p92}. Cycle (5) is well-founded because the size of t (where $t = x\colon t_1 \to t_2$) decreases (to $t_2[(v_1 \underset{r_1}{\overset{r_4}{\bowtie}} t_1)/x]$) when Lemma 15^{p86} calls Lemma 16^{p89}. By the definition of the size of a contract (Figure 6.1), we know that the substitution occuring in t_2 does not change the size of t_2.

Although there are cyclic dependencies among these theorems and lemmas, on each cyclic path, there is a decrease in the size of t. Thus, our proof on induction of the size of t is well-founded.

6.5 Congruence of Crashes-More-Often

Theorem 10 (Congruence of Crashes-More-Often)

$$\forall e_1, e_2.\quad e_1 \preceq e_2 \quad\Longleftrightarrow\quad \forall \mathcal{C},\ \ \mathcal{C}[\![e_1]\!] \preceq \mathcal{C}[\![e_2]\!]$$

Proof We prove two directions separately:

($\Rightarrow$) For an arbitrary $\mathcal{B}$, we prove $\mathcal{B}[\![e_1]\!] \preceq \mathcal{B}[\![e_2]\!]$. We have the following proof:

$$\begin{array}{rl} & e_1 \preceq e_2 \\ \Longleftrightarrow & \text{(By definition 7)} \\ & \forall \mathcal{C}.\mathcal{C}[\![e_2]\!] \rightarrow^* \texttt{BAD} \Rightarrow \mathcal{C}[\![e_1]\!] \rightarrow^* \texttt{BAD} \\ \Rightarrow & \forall \mathcal{C}, \mathcal{D}.\ (\mathcal{C} = \mathcal{D}[\![\mathcal{B}[\![\bullet]\!]]\!]) \Rightarrow (\mathcal{C}[\![e_2]\!] \rightarrow^* \texttt{BAD} \Rightarrow \mathcal{C}[\![e_1]\!] \rightarrow^* \texttt{BAD}) \\ \Rightarrow & \forall \mathcal{D}.\ \mathcal{D}[\![\mathcal{B}[\![e_2]\!]]\!] \rightarrow^* \texttt{BAD} \Rightarrow \mathcal{D}[\![\mathcal{B}[\![e_1]\!]]\!] \rightarrow^* \texttt{BAD} \\ \Rightarrow & \forall \mathcal{B}.\mathcal{B}[\![e_1]\!] \preceq \mathcal{B}[\![e_2]\!] \end{array}$$

Note that we assume for all $i = 1, 2$:

$$\vdash \mathcal{C}[\![e_i]\!] :: (), \vdash \mathcal{D}[\![e_i]\!] :: () \text{ and } \vdash \mathcal{E}[\![e_i]\!] :: ()$$

($\Leftarrow$) It is trivially true, because we can choose an empty context (i.e. $\mathcal{C} = \bullet$). ∎

6.6 Projection Pair and Closure Pair

Recall the definition of projection pair. Let D and E be complete partial order's. If $f : D \rightarrow E$ and $g : E \rightarrow D$ are continuous functions such that $f \circ g \subseteq id$, then (f,g) is called a projection pair. If $id \subseteq f \circ g$, then (f,g) is called a closure pair. In this section, we are not going to explore the theory in depth. We only notice that in some way ($\bullet \triangleright t \triangleleft t \preceq id$) and ($id \preceq \bullet \triangleleft t \triangleright t$) match the definition of projection pair and closure pair respectively.

Theorem 11 (A Projection Pair) *For all e and t, such that $\exists\Delta.\ \Delta \vdash e :: \tau$ and $\Delta \vdash_c t :: \tau$,*

$$(e \triangleright t) \triangleleft t \preceq e$$

Proof We have the following proof:

$$\begin{array}{rl} & (e \triangleright t) \triangleleft t \\ = & \text{(By defn of } \triangleright \text{ and } \triangleleft) \\ & (e \overset{\texttt{BAD}}{\underset{\texttt{UNR}}{\bowtie}} t) \overset{\texttt{UNR}}{\underset{\texttt{BAD}}{\bowtie}} t \\ = & \text{(By Lemma 15}^{\text{p86}}\text{)} \\ & e \overset{\texttt{BAD}}{\underset{\texttt{BAD}}{\bowtie}} t \\ \ll_{\{\texttt{BAD}\}} & e \end{array}$$

By definition of $\ll_{\{\texttt{BAD}\}}$, we get the desired result. ∎

Theorem 12 (A Closure Pair) *For all e and t, such that $\exists\Delta.\ \Delta \vdash e :: \tau$ and $\Delta \vdash_c t :: \tau$,*

$$e \preceq (e \triangleleft t) \triangleright t$$

PROOF We have the following proof:

$$\begin{array}{lll} & (e \lhd t) \rhd t & \\ = & \text{(By defn of } \lhd \text{ and } \rhd) & \\ & (e \underset{\text{BAD}}{\overset{\text{UNR}}{\bowtie}} t) \underset{\text{UNR}}{\overset{\text{BAD}}{\bowtie}} t & \\ = & \text{(By Lemma 15}^{\text{p86}}\text{)} & \\ & e \underset{\text{UNR}}{\overset{\text{UNR}}{\bowtie}} t & \\ \ll_{\{\texttt{UNR}\}} & e & \end{array}$$

By definition of $\ll_{\{\texttt{UNR}\}}$, we get the desired result. ∎

6.7 Contracts are Projections

Recall the definition of projection, a projection p is a function that has two properties:

1. $p = p \circ p$
2. $p \subseteq 1$

The first one is called the retract property and says that projections are idempotent on their range. The second one says that the result of a projection contains no more information than its input.

We would like to show that if $e \in t$, then $(\bullet \lhd t)$ is an error projection while $(\bullet \rhd t)$ is a safe projection. By *error projection*, we mean $e \lhd t$ either behaves the same as e or returns `BAD`. Similarly, by *safe projection*, we mean $e \rhd t$ either behaves the same as e or returns `UNR`.

Findler and Blume [FB06] are the first to discover that contracts are pairs of projections. However, they assume that the e is a non-crashing term and the only error raised are contract violations. We assume that a program may contain errors and may crash. We give *error* a contract `Any`. Moreover, we prove different theorems from [FB06].

Theorem 13 (Error Projection) *For all closed e and closed t, if $e \in t$, $(\bullet \lhd t)$ is a projection.*

PROOF By Lemma 20[p95] (a) (Idempotency) and Lemma 21[p96] (a). ∎

Theorem 14 (Safe Projection) *For all closed e and closed t, if $e \in t$, $(\bullet \rhd t)$ is a projection.*

PROOF By Lemma 20[p95] (b) (Idempotency) and Lemma 21[p96] (b). ∎

Lemma 20 (Idempotence) *For all closed e, t,*

$$e \underset{r_2}{\overset{r_1}{\bowtie}} t \underset{r_2}{\overset{r_1}{\bowtie}} t = e \underset{r_2}{\overset{r_1}{\bowtie}} t$$

PROOF It follows directly from Lemma 15[p86] (telescoping property). ∎

Lemma 21 (Conditional Projection) *For all closed e, closed and crash-free t, if $e \in t$, then*

$$(a) \quad e \triangleleft t \preceq e \qquad\qquad (b) \quad e \preceq e \triangleright t$$

PROOF We prove each of them separately.

(a) Given $e \in t$, we have:

$$\begin{array}{ll} & e \triangleleft t \\ = & \text{(By defn of } \triangleright \text{ in Figure 5.1)} \\ & e \underset{\text{BAD}}{\overset{\text{UNR}}{\bowtie}} t \\ \equiv_s & \text{(By Lemma 22[p96] (Exception III))} \\ & e \underset{\text{BAD}}{\overset{\text{BAD}}{\bowtie}} t \\ \preceq & e \end{array}$$

(b) Given $e \in t$, we have:

$$\begin{array}{ll} & e \triangleright t \\ = & \text{(By defn of } \triangleright \text{ in Figure 5.1)} \\ & e \underset{\text{UNR}}{\overset{\text{BAD}}{\bowtie}} t \\ \equiv_s & \text{(By Lemma 22[p96] (Exception III))} \\ & e \underset{\text{UNR}}{\overset{\text{UNR}}{\bowtie}} t \\ \succeq & \text{(Since } t \text{ is crash-free)} \\ & e \end{array}$$

End of proof. ∎

Lemma 22 (Exception III) $\forall e, t.\ e \in t \Rightarrow \forall r.\ e \underset{r}{\overset{\text{BAD}}{\bowtie}} t \equiv_s e \underset{r}{\overset{\text{UNR}}{\bowtie}} t$

PROOF For all expression e, contract t, we have:

$$\begin{array}{ll} & e \in t \\ \iff & \text{(By Theorem 9[p81] (Grand Theorem))} \\ & e \triangleright t \text{ is } \mathbf{cf} \\ \iff & \text{(By defn of } \triangleright \text{ and } \mathbf{cf}) \\ & \forall \mathcal{C},\ \texttt{BAD} \notin \mathcal{C}.\ \mathcal{C}[\![e \underset{r}{\overset{\text{BAD}}{\bowtie}} t]\!] \not\rightarrow^* \texttt{BAD} \\ \iff & \text{(By Lemma 17[p91] (Exception I))} \\ & \forall \mathcal{C},\ \texttt{BAD} \notin \mathcal{C}.\ \mathcal{C}[\![e \underset{r}{\overset{\text{BAD}}{\bowtie}} t]\!] \equiv_s \mathcal{C}[\![e \underset{r}{\overset{\text{UNR}}{\bowtie}} t]\!] \\ \Rightarrow & \text{(Let } \mathcal{C} = \bullet) \\ & e \underset{r}{\overset{\text{BAD}}{\bowtie}} t \equiv_s e \underset{r}{\overset{\text{UNR}}{\bowtie}} t \end{array}$$

We are done. ∎

6.8 Behaviour of Projections

We have seen that in Section 6.7, we make use of the property of behaves-the-same ($\ll$) (Lemma 23[p97]). In this section, we give its detailed proof. Lemma 23[p97] says that an expression wrapped with a contract behaves either the same as the original expression or returns one of the exceptions which can be either `BAD` or `UNR`.

Lemma 23 (Behaviour of Projection) *For all r_1, r_2, e, crash-free t, such that $\vdash e :: \tau$ and $\vdash_c t :: \tau$, and $r_1, r_2 \in \{\texttt{BAD}, \texttt{UNR}\}$,*

$$e \overset{r_1}{\underset{r_2}{\bowtie}} t \ll_{\{r_1,r_2\}} e$$

Proof The proof begins by dealing with two special cases: $e\uparrow$, $e \to^* \texttt{BAD}$. In both cases, by Definition of $\bowtie$ and Lemma 14[p67] (properties of ‘`seq`‘), we know $e \overset{r_1}{\underset{r_2}{\bowtie}} t \equiv_s e$ and we are done.

Hence, for the rest of the proof we assume that $e \to^* val \notin \{\texttt{BAD}, \texttt{UNR}\}$. We prove it by induction on the size of t. Let R be $\{r_1, r_2\}$.

- t is $\{x \mid p\}$: we have

$$\begin{array}{rcl} e \overset{r_1}{\underset{r_2}{\bowtie}} \{x \mid p\} & = & e \text{ ‘}\texttt{seq}\text{‘ } \texttt{case fin } p[e/x] \texttt{ of} \\ & & \qquad \texttt{True} \to e \\ & & \qquad \texttt{False} \to r_1 \end{array}$$

  Since t is crash-free, $p[e/x] \not\to^* \texttt{BAD}$. So there are two cases to consider:

  - If $p[e/x] \to^* \{\texttt{False}\}$, then $e \overset{r_1}{\underset{r_2}{\bowtie}} \{x \mid p\} \to^* \{r_1\}$ and we are done.
  - If $p[e/x]\uparrow$ or $p[e/x] \to^* \{\texttt{True}\}$, since `fin` converts divergence to `True`, we know `fin` $p[e/x]$ gives `True`. Thus, $e \overset{r}{\bowtie} \{x \mid p\} \to^* \{e\}$ and we are done.

- t is $x\colon t_1 \to t_2$: We have

$$\begin{array}{rcl} e \overset{r_1}{\underset{r_2}{\bowtie}} x\colon t_1 \to t_2 & = & e \text{ ‘}\texttt{seq}\text{‘} \\ & & \lambda v.\ ((e\ (v \overset{r_2}{\underset{r_1}{\bowtie}} t_1)) \overset{r_1}{\underset{r_2}{\bowtie}} t_2[(v \overset{r_2}{\underset{r_1}{\bowtie}} t_1)/x]) \end{array}$$

  Since $e \to^* val \notin \{\texttt{BAD}, \texttt{UNR}\}$, $e \to^* \lambda x.e'$ and $(e \overset{r_1}{\underset{r_2}{\bowtie}} x\colon t_1 \to t_2) \to^* \lambda v.\ ((e\ (v \overset{r_2}{\underset{r_1}{\bowtie}} t_1)) \overset{r_1}{\underset{r_2}{\bowtie}} t_2[(v \overset{r_2}{\underset{r_1}{\bowtie}} t_1)/x])$.

  We want to show that $\forall \mathcal{C}.\ \mathcal{C}[\![e]\!] \to^* r \in R \Rightarrow \mathcal{C}[\![\lambda v.\ ((e\ (v \overset{r_2}{\underset{r_1}{\bowtie}} t_1)) \overset{r_1}{\underset{r_2}{\bowtie}} t_2[(v \overset{r_2}{\underset{r_1}{\bowtie}} t_1)/x]]\!]) \to^* r$. We prove it by induction on strict contexts first and by Lemma 2[p44] (strict context), it is true for all contexts. There are 3 cases to consider:

  1. $\mathcal{C} = [\![\bullet]\!]$;
  2. $\mathcal{C} = \mathcal{D}[\![\texttt{case } \bullet \texttt{ of } alts]\!]$;

3. $\mathcal{C} = \mathcal{D}[\![\bullet\ e_3]\!]$.

Case 1 and 2 are trivially true by inspecting the operational semantics of `'seq'`and `case`. For Case 3, since we prove it by induction on the size of context, we have the following induction hypothesis:

$$\forall \mathcal{D}[\![e]\!] \to^* r \Rightarrow \mathcal{D}[\![\bullet\ e_3]\!] \to^* r \qquad \text{[IH]}$$

So all we need to prove is that for all e_3,

$$(\lambda v.\ ((e\ (v \overset{r_2}{\underset{r_1}{\bowtie}} t_1)) \overset{r_1}{\underset{r_2}{\bowtie}} t_2[(v \overset{r_2}{\underset{r_1}{\bowtie}} t_1)/x]))\ e_3 \ll_R e\ e_3$$

By β-reduction, it means we want to show

$$(e\ (e_3 \overset{r_2}{\underset{r_1}{\bowtie}} t_1)) \overset{r_1}{\underset{r_2}{\bowtie}} t_2[(e_3 \overset{r_2}{\underset{r_1}{\bowtie}} t_1)/x] \ll_R (e\ e_3) \qquad (*)$$

By induction hypotheis where $t = t_2[(e_3 \overset{r_2}{\underset{r_1}{\bowtie}} t_1)/x]$, we have

$$(e\ (e_3 \overset{r_2}{\underset{r_1}{\bowtie}} t_1)) \overset{r_1}{\underset{r_2}{\bowtie}} t_2[(e_3 \overset{r_2}{\underset{r_1}{\bowtie}} t_1)/x] \ll_R (e\ (e_3 \overset{r_2}{\underset{r_1}{\bowtie}} t_1)) \quad (1)$$

By induction hypothesis where $t = t_1$, we have

$$e_3 \overset{r_2}{\underset{r_1}{\bowtie}} t_1 \ll_R e_3$$

By Lemma 24[p98] (Congruence of $\ll_R$), we have

$$e\ (e_3 \overset{r_2}{\underset{r_1}{\bowtie}} t_1) \ll_R e\ e_3 \qquad (2)$$

By (1) and (2) and Lemma 25[p99] (Transitivity of $\ll_R$), we get (*). By [IH], we have the desired result $\forall \mathcal{C}.\ \mathcal{C}[\![e]\!] \to^* r \in R \Rightarrow \mathcal{C}[\![e \overset{r_1}{\underset{r_2}{\bowtie}} x\colon t_1 \to t_2]\!] \to^* r$.

- t is (t_1, t_2): We have

$$\begin{aligned} e \overset{r}{\bowtie} (t_1, t_2) \quad = \quad & \texttt{case}\ e\ \texttt{of} \\ & (e_1, e_2) \to (e_1 \overset{r}{\bowtie} t_1,\ e_2 \overset{r}{\bowtie} t_2) \end{aligned}$$

If $e \to^* val \notin \{\texttt{BAD}, \texttt{UNR}\}$, then $e \to^* \{e_1, e_2\}$. By the induction hypotheses where $t = t_1$ and $t = t_2$ respectively, we know $e_1 \overset{r}{\bowtie} t_1 \ll_R e_1$ and $e_2 \overset{r}{\bowtie} t_2 \ll_R e_2$. Therefore, by Definition 6[p47] (b), we have $e \overset{r}{\bowtie} (t_1, t_2) \ll_R e$.

- t is `Any`: Since we have $e \overset{r}{\bowtie} \texttt{Any} = \neg r$, we know $e \overset{r}{\bowtie} \texttt{Any} \to^* \{\neg r\}$. By Definition 6[p47] (a), we are done. ∎

Lemma 24 (Congruence of Behaves-the-same) *If $e_1 \ll_R e_2$, then $\forall \mathcal{C},\ \mathcal{C}[\![e_1]\!] \ll_R \mathcal{C}[\![e_2]\!]$.*

PROOF we have the following proof:

$$
\begin{array}{ll}
 & e_1 \ll_R e_2 \\
\Longleftrightarrow & \text{(By definition 6)} \\
 & \forall \mathcal{C},\ \mathcal{C}[\![e_2]\!] \to^* r \in R \Rightarrow \mathcal{C}[\![e_1]\!] \to^* r \\
\Rightarrow & \text{(Choose } \mathcal{C} \text{ be } \mathcal{D}[\![\mathcal{C}[\![E]\!]\bullet]\!]) \\
 & \forall \mathcal{D}, \forall \mathcal{E},\ \mathcal{D}[\![\mathcal{E}[\![e_2]\!]]\!] \to^* r \in R \Rightarrow \mathcal{D}[\![\mathcal{E}[\![e_1]\!]]\!] \to^* r \\
\Longleftrightarrow & \text{(By definition 6)} \\
 & \forall \mathcal{C},\ \mathcal{C}[\![e_1]\!] \ll_R \mathcal{C}[\![e_2]\!]
\end{array}
$$

Note that we assume for all $i = 1, 2$:

$$\vdash \mathcal{C}[\![e_i]\!] :: (), \vdash \mathcal{D}[\![e_i]\!] :: () \text{ and } \vdash \mathcal{E}[\![e_i]\!] :: ()$$

End of proof. ∎

Lemma 25 (Transitivity of $\ll_R$) *If $e_1 \ll_R e_2$ and $e_2 \ll_R e_3$, then $e_1 \ll_R e_3$.*

PROOF By Definition 6[p47], we have

$$
\begin{array}{ll}
(1) & \forall \mathcal{C}.\, \mathcal{C}[\![e_2]\!] \to^* r \in R \Rightarrow \mathcal{C}[\![e_1]\!] \to^* r \\
(2) & \forall \mathcal{C}.\, \mathcal{C}[\![e_3]\!] \to^* r \in R \Rightarrow \mathcal{C}[\![e_2]\!] \to^* r
\end{array}
$$

For all $\mathcal{C}$, assuming $\mathcal{C}[\![e_3]\!] \to^* r \in R$, we want to show $\mathcal{C}[\![e_1]\!] \to^* r$. We have the following proof:

$$
\begin{array}{ll}
\forall \mathcal{C}. & \mathcal{C}[\![e_3]\!] \to^* r \in R \\
\Rightarrow & \text{(By (2))} \\
 & \mathcal{C}[\![e_2]\!] \to^* r \in R \\
\Rightarrow & \text{(By (1))} \\
 & \mathcal{C}[\![e_1]\!] \to^* r
\end{array}
$$

End of proof ∎

Chapter 7

Symbolic Execution and Error Reporting

Given $f \in t$, we have shown that to check $f \in t$, we check "$f \triangleright t$ is crash-free" instead. What happens if $f \triangleright t$ is not crash-free? That means when we try to symbolically execute the term $f \triangleright t$ to some e' and there are some residual `BAD`s in e', what can we report to the programmer from the contract violation? In this chapter, we first give the details of the symbolic execution together with a novel counter-example guided (CEG) unrolling approach. Then we are ready to show how meaningful error messages can be generated from the simplified version of $f \triangleright t$.

7.1 Symbolic Execution

After we construct the expression $e \triangleright t$ which captures all contract violations with `BAD`, all we need to do is to symbolically evaluate it and attempt to simplify it to some e' which does not contain `BAD`. Symbolic execution is also called simplification. A set of deterministic simplification rules is shown in Figure 7.2 The function $fv(e)$, which returns free variables of e, is defined in Figure 7.1. Each rule satisfies the theorem:

$$\text{If } e_1 \Longrightarrow e_2, \text{then } e_1 \equiv_s e_2$$

Note that each transition rule in Figure 3.3 itself is also a simplification rule.

7.1.1 Simplification Rules

Many simplification rules are adopted from the literature [Pey96]. For example, the rule CaseCase floats out the scrutinee while the rule CaseOut pushes an argument into each branch. The short-hand $\{K_i\ \vec{x_i} \to e_i\}$ stands for $\forall i, 1 \leq i \leq n.\{K_1\ \vec{x_1} \to e_1; \ldots; K_n\ \vec{x_n} \to e_n\}$ where $\vec{x_i}$ refers to a vector of fields of a constructor K_i. The rest of the rules are elaborated as follows.

$$
\begin{array}{lcl}
fv & :: & \mathbf{Exp} \to \{\mathbf{Var}\} \\
fv(\texttt{BAD}) & = & \emptyset \\
fv(\texttt{UNR}) & = & \emptyset \\
fv(x) & = & \{x\} \\
fv(\lambda x.e) & = & fv(e) - \{x\} \\
fv(e_1\ e_2) & = & fv(e_1) \cup fv(e_2) \\
fv(K\ e_1 \ldots e_n) & = & \bigcup_{i=0}^{n} fv(e_i) \\
fv(\texttt{case}\ e_0\ \{c_i\ \vec{x_i} \to e_i\}) & = & fv(e_0) \cup \bigcup_{i=0}^{n} (fv(e_i) - \vec{x_i})
\end{array}
$$

Figure 7.1: Free Variables

$$
\begin{array}{lll}
 & \texttt{case}\ (\texttt{case}\ e_0\ \texttt{of}\ \{pt_i \to e_i\})\ \texttt{of}\ alts & (\textsc{CaseCase}) \\
\Longrightarrow & \texttt{case}\ e_0\ \texttt{of}\ \{pt_i \to \texttt{case}\ e_i\ \texttt{of}\ alts\} \quad fv(alts) \cap var(pt_i) = \emptyset & \\
 & & \\
 & (\texttt{case}\ e_0\ \texttt{of}\ \{pt_i \to e_i\})\ a & (\textsc{CaseOut}) \\
\Longrightarrow & \texttt{case}\ e_0\ \texttt{of}\ \{pt_i \to (e_i\ a)\} \quad fv(a) \cap \vec{x_i} = \emptyset & \\
 & & \\
 & \texttt{case}\ e_0\ \texttt{of}\ \{pt_i \to e_i; pt_j \to \texttt{UNR}\} & (\textsc{RmUNR}) \\
\Longrightarrow & \texttt{case}\ e_0\ \texttt{of}\ \{pt_i \to e_i\} & \\
 & & \\
 & \texttt{case}\ e_0\ \texttt{of}\ \{pt_i \to e_i\} & (\textsc{SameBranch}) \\
\Longrightarrow & e_0\ \texttt{`seq`}\ e_1 & \\
 & \quad \text{patterns are exhaustive and} & \\
 & \quad \text{for all } i, fv(e_i) \cap var(pt_i) = \emptyset \text{ and } e_1 = e_i & \\
 & & \\
 & \texttt{case}\ e_0\ \texttt{of}\ \{K_i\ \vec{x_i} \to e_i\} & (\textsc{Scrut}) \\
\Longrightarrow & \texttt{case}\ e_0\ \texttt{of}\ \{K_i\ \vec{x_i} \to e_i[K_i\ \vec{x_i}/e_0]\} \qquad fv(e_i) \cap \vec{x_i} = \emptyset &
\end{array}
$$

Figure 7.2: Simplification Rules

Unreachable The rule RmUNR discards all unreachable branches which are marked by `UNR`. For example:

```
... case xs of
      [] -> UNR
      (x:xs') -> x
```

will be simplified to

```
... case xs of
      (x:xs') -> x
```

One difference between the semantics of our language $\mathcal{H}$ and Haskell is that: *in Haskell, a missing branch denotes a crashing branch while in $\mathcal{H}$, a missing branch denotes an unreachable branch.* There are two reasons we design the semantics in this way:

- When we preprocess a user program (i.e. transform Haskell to language $\mathcal{H}$), all missing cases of pattern matching are explicitly replaced by a branch that returns `BAD` as mentioned earlier in Section 3.1.
- If we know a branch is not reachable, we want to delete the branch so as to reduce the size of the expression under symbolic simplification.

For example, we may have a code fragment like:

```
... case False of
      True -> ...
      False -> UNR
```

which can be simplified to:

```
... case False of
      True -> ...
```

We can see that in this case, the scrutinee does not match any branch. Now, we can apply the transition rule [E-Match4] (in Figure 3.3) and simplify the whole case-expression to `UNR`.

Match The transition rule [E-Match4] (in Figure 3.3) and the simplification rule RmUNR might seem to be redundant due to the existence of the transition rules [E-Match1-3], which select the matched branch and remove the unmatched branches. But they are not. Consider:

```
... case xs of
     True -> case False of
               True -> ...
     False -> ...
```

The transition rules [E-Match1-3] only deal with the situation when the scrutinee matches one of the branches. So in the above case, we need to apply the transition rule [E-Match4] and the simplification rule RmUNR respectively to get:

```
... case xs of
     False -> ...
```

Common Branches During the simplification process, we often encounter code fragment like this:

```
... case xs of
     K1 -> True
     K2 -> True
```

In the rule SameBranch if all branches have the identical RHS (w.r.t. α-conversion), the scrutinee is redundant. However, we need to be careful as we should do this *only if*

(a) all patterns are exhaustive (i.e. all constructors of a data type are tested);

(b) given $\{K_i\ \vec{x_i} \rightarrow e_i\}$, no free variables in e_i are bound in $K_i\ \vec{x_i}$.

To see why we need condition (a), consider the following example:

```
rev :: {xs | True} -> {rs | null rs ==> null xs}
```

During the simplification of its checking code `rev` $\triangleright\ t_{rev}$, we may have:

```
... case rev xs of
    [] -> case xs of
            [] -> rev xs
    (x:xs') -> ...
```

The inner `case` has only one branch (the other branch is understood to be unreachable). It might be believed that we would replace the expression `(case xs of {[] -> rs })` by `rs` as there is only one branch that is reachable and the resulting expression does not rely on any substructure of `xs`. However, this makes us lose a critical piece of information, namely:

$$\textit{if}\ \texttt{(rev xs) == []}, \textit{then}\ \texttt{xs == []}.$$

On the other hand, given this information we can perform more aggressive simplification. For example, suppose we have another function `g` that calls `rev`:

```
g xs = case (rev xs) of
         [] -> ... case xs of
                     [] -> True
                     (x:xs) -> False
         (x:xs) -> ...
```

we may use the above information to simplify the inner `case` to `True` which may allow more aggressive symbolic checking.

The condition (b) is needed because of the scoping of variables. For example:

```
case xs of
  K1 x y -> y
  K2 x y -> y
```

The two ys in the two branches refer to different data although they are syntactically the same. So the rule SameBranch can only be applied when e_i does not use any variables in $\vec{x_i}$.

Moreover, the RHS of the rule SameBranch is e_0`` `seq` ``e_1 instead of e_1 is because if e_0 reduces to `BAD` (or `UNR`), the LHS reduces to `BAD` (or `UNR`), which is not semantically equivalent to e_1.

Static Memoization All known information should be used in simplifying an expression. In order for the rule SCRUT to work, we need to keep a table which captures all the information we know when we traverse the syntax tree of an expression. As the scrutinee of a case-expression is an expression, the key of the table is an expression rather than a variable. The value of the table is the information that is true for the corresponding scrutinee. For example, when we encounter:

```
case (noT1 x) of
 True -> e1
```

we extend the information table like this:

:	:
noT1 x	True

When we symbolically evaluate `e1` and encounter `(noT1 x)` a second time in `e1`, we look up its corresponding value in the information table for substitution.

7.1.2 Arithmetic

Our simplification rules are mainly to handle pattern matchings. For expressions involving arithmetic, we need to consult a theorem prover. Suppose we have:

```
foo :: Int -> Int -> Int
foo :: {i | True} -> {j | i > j}
```

Its representative function `foo` $\triangleleft$ $t_{\texttt{foo}}$ looks like this:

```
t_foo = \i j -> case (i > j) of
                  False -> BAD
                  True  -> ...
```

Now, suppose we have a call to `foo`:

```
goo i = foo (i+8) i
```

After inlining `foo` $\triangleleft$ t_{foo}, we may have such symbolic checking code:

```
\i -> case (i+8 > i) of
        False -> BAD
        True  -> ...
```

A key question to ask is if `BAD` can be reached? To reach `BAD`, we need `i+8 > i` to return `False`. Now we can pass this off to a theorem prover that is good at arithmetic and see if we can prove that the `False` branch is unreachable. If so, we can safely remove the branch leading to `BAD`.

In theory, we can use any theorem prover that can perform arithmetic. Currently, we choose a free theorem prover named *Simplify* [DNS05] to perform arithmetic checking in an incremental manner. For each case scrutinee such that

- it is an expression involving solely primitive operators, or
- it returns a boolean data constructor

we invoke *Simplify* prover to determine if this scrutinee evaluates to definitely *true*, definitely *false* or *DontKnow*. If the answer is either *true* or *false*, the simplification rule of MATCH is applied as well as adding this to our information table. Otherwise, we just keep the scrutinee and continue to symbolically evaluate the branches.

Each time we query the theorem prover *Simplify*, we pass the knowledge accumulated in our information table as well. For example, we have the following fragment during the simplification process:

```
... case i > j of
     True -> case j < 0 of
               False -> case i > 0 of    -- (*)
                          False -> BAD
```

When we reach the line marked by `(*)`, before we query `i > 0`, we send information `i > j == True` and `j < 0 == False` to the *Simplify*. Such querying can be efficiently implemented through the push/pop commands supplied by the theorem prover which allows truth information to be pushed to a global (truth) stack and popped out when it is no longer needed.

You might notice that it is possible to make use of the external theorem prover to do the static memoization job mentioned in Section 7.1.1. There are two reasons that we do not combine two tasks into one:

1. calling an external theorem prover is more expensive. Currently, we covert program fragment to *string* format and send it to the external theorem prover; after getting the result (from the prover) which is also in *string* format, we *parse* the string in order to know the result is "Valid" or "Invalid". Thus, it is more expensive than looking up a static memoization table internally. Moreover, if there is no arithmetic involved in a program (eg. programmers use Peano numbers only), programmers may choose not to install the external theorem prover or to switch off the prover by setting a flag during compilation. So we do not have to enforce programmers to install an external prover if programmers work on a platform (e.g. Palm) with limited resources.

2. external stuff is less trustworthy. We have proved the correctness of the whole verification system except the correctness of the external theorem prover because it is developed by another party.

7.1.3 Proof of Soundness of Simplification

In this section, we show that each simplification rule preserves the semantics of the expression it simplifies.

Lemma 26 (Correctness of One-Step Simplification) *If $e_1 \Longrightarrow e_2$, then $e_1 \equiv_s e_2$.*

PROOF We prove the lemma by considering simplifying rules one by one. Basically, we symbolically evaluate both e_1 and e_2. If $\exists e'.\ e_1 \rightarrow^* e'$ and $e_2 \rightarrow^* e'$, then we are done.

- Rule CASEOUT: We want to show $e_1 \equiv_s e_2$ where

$$\begin{array}{rcl} e_1 & = & (\texttt{case}\ e_0\ \texttt{of}\ \{pt_i \rightarrow e_i\})\ a \\ e_2 & = & \texttt{case}\ e_0\ \texttt{of}\ \{pt_i \rightarrow e_i\ a\} \end{array}$$

  There are four cases to consider:

  (a) Case $e_0 \rightarrow^* \texttt{BAD}$. By [E-excase] and [E-exapp], $e_1 \rightarrow^* \texttt{BAD}$. By [E-excase], $e_2 \rightarrow^* \texttt{BAD}$.

  (b) Case $e_0\uparrow$. By [E-excase] and [E-exapp], $e_2\uparrow$. By [E-excase], $e_2\uparrow$.

  (c) Case $e_0 \rightarrow^* K_j\ \vec{x_j}$. By [E-match1], both e_1 and e_2 reduce to $(e_j\ a)$ if we have $\{\ldots; K_j\ \vec{x_j} \rightarrow e_j; \ldots\}$.

  (d) Case $e_0 \rightarrow^* K_j\ \vec{x_j}$ where $K_j\ \vec{x_j} \notin \{pt_1, \ldots, pt_{n-1}\}$. By [E-match2], both e_1 and e_2 reduces to $(e_n\ a)$ if we have $\{\ldots; \texttt{DEFAULT} \rightarrow e_n\}$.

- Rule CASECASE: We want to show $e_1 \equiv_s e_2$ where

$$\begin{array}{rcl} e_1 & = & \texttt{case}\ (\texttt{case}\ e_0\ \texttt{of}\ \{pt_i \rightarrow e_i\})\ \texttt{of}\ alts \\ e_2 & = & \texttt{case}\ e_0\ \texttt{of}\ \{pt_i \rightarrow \texttt{case}\ e_i\ \texttt{of}\ alts\} \end{array}$$

  There are 3 cases to consider:

  (a) Case $e_0 \rightarrow^* \texttt{BAD}$. We have:

$$\begin{array}{ll} e_1 \rightarrow^* \texttt{BAD} & \text{(by applying [E-excase] twice)} \\ e_2 \rightarrow^* \texttt{BAD} & \text{(by [E-excase])} \end{array}$$

  (b) Case $e\uparrow$. We have:

$$\begin{array}{ll} e_1\uparrow & \text{(by applying [E-excase] twice)} \\ e_2\uparrow & \text{(by [E-excase])} \end{array}$$

  (c) Case $e_0 \rightarrow^* pt_j$. We have:

$$\begin{array}{ll} e_1 \rightarrow^* \texttt{case}\ e_j\ \texttt{of}\ alts & \text{(by [E-match1] or [E-match2)} \\ e_2 \rightarrow^* \texttt{case}\ e_j\ \texttt{of}\ alts & \text{(by [E-match1] or [E-match2)} \end{array}$$

- Rule NoMatch: We want to show $e_1 \equiv_s e_2$ where

$$\begin{array}{lcl} e_1 & = & \texttt{case}\ K_j\vec{e_j}\ \texttt{of}\ \{pt_i \to e_i\} \\ e_2 & = & \texttt{UNR} \end{array}$$

  The expression e_1 gets stuck as there is no reduction rule to apply. Since `UNR` denotes getting stuck, we have $e_1 \equiv_s e_2$.

- Rule RmUNR: We want to show $e_1 \equiv_s e_2$ where

$$\begin{array}{lcl} e_1 & = & \texttt{case}\ e_0\ \texttt{of}\ \{K_i\ \vec{x_i} \to e_i; pt_j \to \texttt{UNR}\} \\ e_2 & = & \texttt{case}\ e_0\ \texttt{of}\ \{K_i\ \vec{x_i} \to e_i\} \end{array}$$

  If $e_0 \to^* pt_j$, then $e_1 \to^* \texttt{UNR}$ and e_2 gets stuck. Since `UNR` denotes getting stuck, we have $e_1 \equiv_s e_2$.

- Rule SameBranch: We want to show $e_1 \equiv_s e_2$ where

$$\begin{array}{lcl} e_1 & = & \texttt{case}\ e_0\ \texttt{of}\ \{pt_i \to e_i'\} \\ e_2 & = & e_0\ \texttt{`seq`}\ e_1' \end{array}$$

  There are 3 cases to consider:

  (a) Case $e_0 \to^* \texttt{BAD}$. We have:

$$\begin{array}{ll} e_1 \to^* \texttt{BAD} & \text{(by [E-excase])} \\ e_2 \to^* \texttt{BAD} & \text{(by defn of `seq` and [E-excase])} \end{array}$$

  (b) Case $e_0\uparrow$. We have:

$$\begin{array}{ll} e_1\uparrow & \text{(by [E-excase])} \\ e_2\uparrow & \text{(by defn of `seq` and [E-excase])} \end{array}$$

  (c) Case $e_0 \to^* pt_j$. We have:

$$\begin{array}{ll} e_1 \to^* e_j & \text{(by [E-match1])} \\ e_2 \to^* e_1 & \text{(by defn of `seq` and [E-match2])} \end{array}$$

  Since for all i, $fv(e_i) \cap \vec{x_i} = \emptyset$ and $e_1 = e_i$, we know $e_1 = e_j$.

- Rule Scrut: We want to show $e_1 \equiv_s e_2$ where

$$\begin{array}{lcl} e_1 & = & \texttt{case}\ e_0\ \texttt{of}\ \{K_i\ \vec{x_i} \to e_i\} \\ e_2 & = & \texttt{case}\ e_0\ \texttt{of}\ \{K_i\ \vec{x_i} \to e_i[K_i\ \vec{x_i}/e_0]\} \end{array}$$

  There are 3 cases to consider:

  (a) Case $e_0 \to^* \texttt{BAD}$. We have:

$$\begin{array}{ll} e_1 \to^* \texttt{BAD} & \text{(by [E-excase])} \\ e_2 \to^* \texttt{BAD} & \text{(by [E-excase])} \end{array}$$

(b) Case $e\uparrow$. We have:

$$\begin{array}{ll} e_1\uparrow & \text{(by [E-excase])} \\ e_2\uparrow & \text{(by [E-excase])} \end{array}$$

(c) Case $e_0 \rightarrow^* K_j\ \vec{x_j}$. We have:

$$\begin{array}{ll} e_1 \rightarrow^* e_j & \text{(by [E-match1] or [E-match2])} \\ e_2 \rightarrow^* e_j[K_j\ \vec{x_j}/e_0] & \text{(by [E-match1] or [E-match2])} \end{array}$$

Since $e_0 \rightarrow^* K_j\ \vec{x_j}$, by the confluence of the language, we know $e_0 \equiv_s K_j\ \vec{x_j}$. So $e_j[K_j\ \vec{x_j}/e_0] \equiv_s e_j$. ∎

7.2 Counter-Example Guided Unrolling

If many cases, applying simplification rules to an expression is not good enough to determine whether all `BAD`s are reachable because we may have arbitrary function calls in the expression. Consider:

```
sumT :: T -> Int
sumT :: {x | noT1 x } -> {r | True }
sumT (T2 a) = a
sumT (T3 t1 t2) = sumT t1 + sumT t2
```

where `noT1` is the recursive predicate mentioned in Section 2.1.1. After simplifying $\texttt{sumT}\triangleright t_{\texttt{sumT}}$, we may have:

```
case (noT1 x) of
 True -> case x of
          T1 a -> BAD
          T2 a -> a
          T3 t1 t2 -> case (noT1 t1) of
                       False -> BAD
                       True -> case (noT1 t2) of
                                False -> BAD
                                True -> sumT t1 + sumT t2
```

Program Slicing To focus on our goal (i.e. removing `BAD`s) as well as to make the checking process more efficient, we slice the program by collecting only the paths that lead to `BAD`. We assume all variables are safe. A bound variable is safe because we want to assume a safe argument will be given. A variable that denotes a top-level function is safe because we already wrap this variable with its contract. That means all potential crashes are exposed in this wrapping. A function named `slice`, which does the job, is defined in Figure 7.3. A call to `slice` gives the following sliced program:

```
case (noT1 x) of
 True -> case x of
          T1 a -> BAD
          T3 t1 t2 -> case (noT1 t1) of
                        False -> BAD
                        True -> case (noT1 t2) of
                                  False -> BAD
```

In Figure 7.3, the function `checkAndSlice` filters all `BAD` branches when traversing the definition of a function. The list of variables vs captures all local variables: the parameters of the function and the bounded variables in each alt. Function `checkAndSlice` returns two results: the first one is the sliced function definition while the second result indicates there is any `BAD` in the expression under inspection. So in the definition of `noBAD`, we reuse the function `checkAndSlice` as shown in Figure 7.5. The special constructor `Inside` is for tracing which function calls lead to the crash. To save the trouble of having a separate figure for defining `checkAndSlice` and `unroll`, we include their definitions for the constructor `Inside` in Figure 7.3 and Figure 7.4 respectively. Details on `Inside` is in Section 7.3.

$$
\begin{array}{lcl}
\texttt{slice} & :: & \textbf{Exp} \rightarrow \textbf{Exp} \\
\texttt{slice}\ e & = & \texttt{fst}\ (\texttt{checkAndSlice}\ [\,]\ e) \\
 & & \\
\texttt{checkAndSlice}\ vs\ \texttt{BAD} & = & (\texttt{BAD}, \texttt{False}) \\
\texttt{checkAndSlice}\ vs\ \texttt{UNR} & = & (\texttt{UNR}, \texttt{True}) \\
\texttt{checkAndSlice}\ vs\ (v) & = & (v, \texttt{True}) \\
\texttt{checkAndSlice}\ vs\ (e_1\ e_2) & = & \texttt{let}\ (a_1, b_1) = (\texttt{checkAndSlice}\ vs\ e_1) \\
 & & \qquad (a_2, b_2) = (\texttt{checkAndSlice}\ vs\ e_2) \\
 & & \texttt{in if}\ b_1\ \texttt{then if}\ b_2\ \texttt{then}\ (e_1\ e_2, \texttt{True}) \\
 & & \qquad\qquad\qquad\qquad \texttt{else}(e_1\ a_2, \texttt{False}) \\
 & & \qquad \texttt{else}\ (a_1\ a_2, \texttt{False}) \\
\texttt{checkAndSlice}\ vs\ (\lambda x.e) & = & \texttt{let}\ (a, b) = \texttt{checkAndSlice}\ vs\ e \\
 & & \texttt{in if}\ b\ \texttt{then}\ (\lambda x.e, \texttt{True}) \\
 & & \qquad \texttt{else}\ (\lambda x.a, \texttt{False}) \\
\texttt{checkAndSlice}\ vs\ (K\ \vec{e}) & = & \texttt{let}\ es = (map\ (\texttt{checkAndSlice}\ vs)\ \vec{e})) \\
 & & \texttt{in if}\ all\ (map\ (==\texttt{UNR})\ s) \\
 & & \qquad \texttt{then}\ (K\ \vec{e}, \texttt{True}) \\
 & & \qquad \texttt{else}\ (K\ (map\ fst\ es), \texttt{False}) \\
\texttt{checkAndSlice}\ vs\ (\texttt{Inside}\ n\ e) & = & \texttt{let}\ (a, b) = (\texttt{checkAndSlice}\ vs\ e) \\
 & & \texttt{in if}\ b\ \texttt{then}\ (\texttt{Inside}\ n\ e, \texttt{True}) \\
 & & \qquad \texttt{else}\ (\texttt{Inside}\ n\ a, \texttt{False}) \\
\texttt{checkAndSlice}\ vs\ (\texttt{case}\ e_0\ \texttt{of}\ alts) & = & \\
\multicolumn{3}{l}{\quad \texttt{case}\ e_0\ \texttt{of}\ (filter\ (\lambda(K\ \vec{x}\ e) \rightarrow not\ (snd\ (\texttt{checkAndSlice}\ vs\ (e)))\ alts)}
\end{array}
$$

Figure 7.3: Slicing

The Unrolling Itself We know we need to unroll one or all of the call(s) to `noT1` in order to proceed. Let us unroll them one by one. The unrolling is done by a function named `unroll` which is defined in Figure 7.4. The unrolling of the topmost `noT1` gives:

```
case (\x -> case x of
              T1 a' -> False
              T2 a' -> True
              T3 t1' t2' -> noT1 t1' && noT1 t2') x) of
 True -> case x of
           T1 a -> BAD
           T3 t1 t2 -> case (noT1 t1) of
                         False -> BAD
                         True -> case (noT1 t2) of
                                   False -> BAD
```

The function `unroll` takes an expression to be unrolled, an environment ρ that maps top-level function name to its definition and returns a new expression with top-level functions being unrolled. The $\rho(v)$ looks up the environment ρ with v as a key and fetches the definition of v if v is in the environment; otherwise, returns v.

$$
\begin{array}{lcl}
\texttt{unroll} & :: & \mathbf{Exp} \to [(\mathbf{Name}, \mathbf{Exp})] \to \mathbf{Exp} \\
\texttt{unroll}\ (e_1\ e_2)\ \rho & = & ((\texttt{unroll}\ e_1\ \rho)\ e2) \\
\texttt{unroll}\ (v)\ \rho & = & \rho(v) \\
\texttt{unroll}\ (\lambda x.e)\ \rho & = & \lambda x.(\texttt{unroll}\ e\ \rho) \\
\texttt{unroll}\ (K\ e_1 \ldots e_n)\ \rho & = & K\ (\texttt{unroll}\ e_1\ \rho) \ldots (\texttt{unroll}\ e_n\ \rho) \\
\texttt{unroll Inside}\ n\ e & = & \texttt{Inside}\ n\ (\texttt{unroll}\ e\ \rho) \\
\texttt{unroll}\ (\texttt{NoInline}\ e)\ \rho & = & \texttt{NoInline}\ e \\
\multicolumn{3}{l}{\texttt{unroll}\ (\texttt{case}\ e_0\ \texttt{of}\ \{K_i\ \vec{x_i} \to e_i\})\ \rho =} \\
\multicolumn{3}{l}{\quad \texttt{case}\ (\texttt{case}\ (\texttt{unroll}\ e_0\ \rho)\ \texttt{of}\ \{K_i\ \vec{x_i} \to \texttt{NoInline}\ e_0\})\ \texttt{of}} \\
\multicolumn{3}{l}{\qquad \{K_i\ \vec{x_i} \to \texttt{unroll}\ e_i\ \rho\}\}}
\end{array}
$$

Figure 7.4: Unrolling

Keeping Known Information Note that the new information `noT1 t1' && noT1 t2'` after the unrolling is what we need to prove that `(noT1 t1)` and `(noT1 t2)` are not `False` at the branches. However, if we also unroll the calls `(noT1 t1)` and `(noT1 t2)` at the branches, we lose the information `(noT1 t1) == False` and `(noT1 t2) == False`. To solve this problem (i.e. to keep known information), we add one extra case-expression after each unrolling. So unrolling the call of `(noT1 x)` actually yields:

```
case (case (NoInline (noT1 x)) of
       True -> (\x -> case x of
                        T1 a' -> False
                        T2 a' -> a'
                        T3 t1' t2' -> (noT1 t1' && noT1 t2'))) x) of
```

```
True -> case x of
          T1 a -> BAD
          T3 t1 t2 -> case (noT1 t1) of
                        False -> BAD
                        True -> case (noT1 t2) of
                                  False -> BAD
```

But to avoid unrolling the same call more than once, we wrap `(noT1 x)` with a special constructor `NoInline`, which prevents the function `unroll` from unrolling it again.

Counter-Example Guided Unrolling - The Algorithm Given a checking code $f \triangleright t_f$, as we have seen, in order to remove `BAD`s, we may have to unroll some function calls. The counter-example guided unrolling technique is summarised by the pseudo-code algorithm `escH` defined below. The parameter n is the maximum number of unrollings we do. It can be pre-set by either the system or programmers.

```
escH rhs 0 = "Counter-example :" ++ report rhs
escH rhs n =  let rhs' = simplifier rhs
                  b = noBAD rhs'
              in case b of
                  True → "No  Bug."
                  False →  let s = slice rhs'
                           in case noFunCall s of
                               True →  let eg = oneEg s
                                       in "Definite Bug :" ++ report eg
                               False →  let s' = unrollCalls s
                                        in escH s' (n − 1)
```

Basically, the `escH` function simplifies the code $f \triangleright t_f$ and checks that all `BAD`s are removed by the simplification process. If there is any residual `BAD`, it will report to the programmer by generating a warning message. To guarantee termination, `escH` takes a pre-set number which indicates the maximum unrolling that should be performed. Before this number decreases to 0, it simplifies the code $f \triangleright t_f$ once and calls `noBAD`, which is defined in Figure 7.5, to check for the absence of `BAD`. If there is any `BAD` left, it slices the rhs' and obtain an expression which contains all paths that lead to `BAD`. If there is no function calls in the sliced expression which can be checked by a function named `noFunCalls`, we know the existence of a definite bug and report it to programmers. In our system, programmers can pre-set an upper bound on the number of counter-examples that will be generated for the contract checking of each function. By default, it gives one counter-example. If there are function calls, we unroll each of them by calling `unroll`.

This procedure is repeated until either all `BAD`s are removed or the pre-set number of unrollings has decreased to 0. When `escH` terminates, there are three possible outcomes:

- No `BAD` in the resulting expression (which implies definitely safe);
- `BAD` appears and there is no function calls in the resulting expression (where each such `BAD` implies a contract failure);

- `BAD` appears and there are function calls in the resulting expression (where each such `BAD` implies a possible contract failure).

These are essentially the three types of messages we suggest to report to programmers in Section 2.2.

```
noBAD    :: Exp → Bool
noBAD e  =  snd (checkAndSlice [ ] e)
```

Figure 7.5: Checking for `BAD`

From our experience, unrolling is used in the following situations:

1. A recursive predicate (say `noT1`) is used in the contract of another function (say `sumT1`). During the checking process, only the recursive predicates are unrolled. We do not need to unroll `sumT1` at all as its recursive call is represented by its contract whose information is enough for the checking to be done. Thus, we recommend programmers to use recursive predicate of small code size.

2. A function without contract annotation. If it is a recursive function, sometimes we may have to unroll its recursive call to obtain more information during checking. An example is illustrated in Section 8.4.

3. The contract itself does not capture enough information. For example, the function `length`:

   ```
   {-# CONTRACT length :: {xs | True} -> {r | r >= 0} #-}
   length []     = 0
   length (x:xs) = 1 + length xs
   ```

   The contract only specifies that the result should be a non-negative integer. When we encounter a function call `(length [])`, we may have to inline the definition of `length` to know `(length [])` is 0. In this case, there is no contract that is more compact than the function definition itself.

7.2.1 Inlining Strategies

Recall that during the counter-example-guided (CEG) unrolling, whenever we unroll a function call, after inlining the definition of the function, we also keep the call itself using a special constructor `NoInline`. For example, we have:

```
f x   = g x 1
g x y = x > y
```

Suppose we need to CEG unroll the following fragment:

```
case (f a) of
 True -> case (g a 1) of
          False -> BAD
```

we have:

```
case (g a 1) of
 True -> case (NoInline (f x)) of
          True -> case (a > 1) of
                   False -> case (NoInline (g a 1)) of
                             False -> BAD
```

In this case, since we know `(g a 1)` is `True`, the `error` branch is not reachable. As we inline all function calls simultaneously, we need to keep each original call so that we will not lose any known information.

However, this unrolling strategy is not good enough because sometimes we do not want to inline all function calls in a scrutinee at the same time. For example:

```
case (length b + length (a:y) == r) of
 True -> case (length (a:b) + length y == r) of
          False -> BAD
```

We would like to expand the calls `length (a:y)` and `length (a:b)`, but not the calls `length b` and `length y` so that we can get a simplified code like:

```
case (length b + 1 + length y == r) of
 True -> case (1 + length b + length y == r) of
          False -> BAD
```

Now we can hand it over to an external theorem prover to deal with the commutativity of the `+` and show that the `BAD` is unreachable.

To know which call to unroll, there may be a number of strategies. One of them could be *unrolling those calls such that some of their arguments are constructors*. However, this is only reasonable if the function definition pattern matches on those parameter. Our current approach is to unroll all functions except `length` simultaneously and we only unroll the function `length` if its argument is a constructor. With this simple heuristic added in, we can verify properties involving `length` more efficiently. We can add more and more heuristics for other frequently used functions gradually.

Another more general approach could be introducing a special form, `%` which captures both the original call and the inlined expression. Consider the previous example, we may get something like this:

```
case ((length b) % (case b of ...) +
      (length (a:y)) % (case (a:y) of ...) == r) of
 True -> case ((length (a:b)) % (case (a:b) of ...) +
               (length y) % (case y of ...) == r) of
          False -> BAD
```

The LHS of `%` is the original call while the RHS is the inlined body. It means we may have to try all combinations when we test the reachability of the `BAD` branch. This is the drawback of the approach, but it is the most general one which works for all cases.

7.3 Error Tracing and Counter-Example Generation

To know which function to blame [FF02], we need to give each `BAD` a tag. That is `BAD` *lbl* where the label *lbl* is a function name. For a function f with contract t, we check $f \triangleright t$ is crash-free or not, where

$$f \triangleright t = f \overset{\text{BAD "f"}}{\underset{\text{UNR}}{\bowtie}} t$$

A residual `BAD "f"` tells us that we should blame f. That means even if f takes arguments satisfying their corresponding preconditions, f may fail to produce a result that meets its postcondition. For example:

```
{-# CONTRACT inc :: {x | x > 0} -> {r | r > x} #-}
inc x = x - 1
```

after optimizing (i.e. simplifying) $\texttt{inc} \triangleright t_{\texttt{inc}}$, we have:

```
\v -> (case v > 0 of
         True -> case v - 1 > v of
                   True -> case v > 0 of
                             True -> v - 1
                             False -> UNR
                   False -> BAD "inc"
         False -> UNR)
```

We can report to programmers during compile-time:

```
Error: inc fails its postcondition
       when v > 0 holds
            v - 1 > v does not hold
```

This error message is generated directly from the path that leads to the `BAD "inc"`.

```
case v > 0 of
  True -> case v - 1 > v of
            False -> BAD "inc"
```

How about precondition violation? Suppose we know $f \in t_f$. If f is called in a function g with contract t_g, recalling the reasoning in Section 4.4, we shall check $\lambda f.e_g \in t_f \to t_g$. That means we check: $\lambda f.e_g \triangleright t_f \to t_g$ is crash-free or not. By the definition of $\triangleright$, we have: $\lambda f.\ ((e_g\ (f \triangleleft t_f)) \triangleright t_g)$.

In order to trace which function calls which function that fails which function's precondition, instead of using $(f \triangleleft t_f)$, we use:

$$\texttt{Inside}\ lbl\ loc\ (f \triangleleft t_f)$$

The *lbl* is the function name ("f" in this case) and the *loc* indicates the location (e.g. (row,column)) of the definition of f in the source file. Note that

$$f \triangleleft t = f \underset{\text{BAD "f"}}{\overset{\text{UNR}}{\bowtie}} t$$

For example, we have:

```
{-# CONTRACT f1 :: {x | True} -> {z | x < z}
                -> {r | True} #-}
f1 x z = if x < z then 5 else error "Urk"
f2 x z = 1 + f1 x z

f3 [] z = 0
f3 (x:xs) z = case x > z of
               True -> f2 x z
               False -> ...
```

After optimizing $\texttt{f3} \triangleright t_{\texttt{f3}}$, we have:

```
\xs -> \z ->
 case xs of
 [] -> 0
 (x:y) -> case x > z of
          True -> Inside "f2" <l2>
                  (Inside "f1" <l1> (BAD "f1"))
          False -> ...
```

Note that, the `"f1"` in `(BAD "f1")` indicates which function's precondition is not fulfilled. Thus, the residual fragment enables us to give one counter-example with the following meaningful message at compile-time:

```
Error <l3>: f3 (x:y) z
              when x > z holds
              calls f2
              which calls f1
              which may fail f1's precondition!
```

where the location `<l3>` indicates the location of the definition of `f3` in the source file.

Simplification rules related to `Inside` are shown in Figure 7.6. The basic idea is to push `Inside` into substructures of each expression. Moreover, if the expression is a constant, we can remove `Inside` by this rule: $\texttt{Inside}\ f\ l\ n \implies n$ (RmInside)

This error tracing technique achieves the same goal as that in [MFF06] but in a much simpler way.

$$\texttt{Inside}\ f\ l\ (\texttt{case}\ e_0\ \texttt{of}\ \{c_i\ \vec{x_i} \to e_i\})$$
$$\Longrightarrow \texttt{case}\ (\texttt{Inside}\ f\ l\ e_0)\ \texttt{of}\ \{c_i\ \vec{x_i} \to \texttt{Inside}\ f\ l\ e_i\} \qquad \text{(InsideCase)}$$

$$\texttt{Inside}\ f\ l\ (\lambda x.e) \Longrightarrow (\lambda x.\texttt{Inside}\ f\ l\ e) \qquad \text{(InsideLam)}$$

$$\texttt{Inside}\ f\ l\ (e_1\ e_2) \Longrightarrow (\texttt{Inside}\ f\ l\ e_1)\ (\texttt{Inside}\ f\ l\ e_2) \qquad \text{(InsideApp)}$$

$$\texttt{Inside}\ f\ l\ (K\ x_1, \ldots, x_n)$$
$$\Longrightarrow (K\ (\texttt{Inside}\ f\ l\ x_1)\ (\texttt{Inside}\ f\ l\ x_n)) \qquad \text{(InsideConstructor)}$$

Figure 7.6: Simplification Rules for Tracing

Chapter 8

Examples

In this section, we illustrate the power and the limitations of our system with examples. Examples with recursive functions called in contracts are worked out by hand (Section 8.2, 8.3, 8.5 and 8.5) while the rest (Section 8.1 and 8.4) can be automated. This is due to the fact that we do not have a smart algorithm knowing which call to unroll although it is obvious to a normal programmer (a human being) (Section 7.2.1).

8.1 Nested Recursion

The McCarthy's `f91` function always returns 91 when its given input is less than or equal to 101. We can specify this property by the following contract that can be automatically checked.

```
{-# CONTRACT f91 :: {n | True} -> {r | ((n <= 100 && r == 91) ||
                                        n > 100 && r == n - 10)} #-}
f91 :: Int -> Int
f91 n = case (n <= 100) of
          True -> f91 (f91 (n + 11))
         False -> n - 10
```

This example shows how contracts can be exploited to give succinct and precise abstraction for functions with complex recursion.

8.2 Size of Data Structure

These examples show that we can specify and verify size properties of a function in its contract. We first give the function `length` a contract stating that it always returns a non-negative number. We also define a predicate `sameLen` that tests whether two lists have the same length. The difference between `sameLen xs ys` and `length xs == length ys` is that if any of the list `xs` or `ys` is infinite, the test `sameLen xs ys` terminates while the test `length xs == length ys` diverges.

```
{-# CONTRACT length :: Ok -> {r | r >= 0} #-}
length []     = 0
length (x:xs) = 1 + length xs

sameLen [] []         = True
sameLen (x:xs) (y:ys) = sameLen xs ys
sameLen _ _           = False
```

Consider two versions of `reverse`: `reverse1` makes use of recursion to reverse a list while `reverse2` calls an auxiliary function `rev` which makes use of iteration to accumulate resulting list with its second parameter. Our framework can verify that `reverse1`, `rev` and `reverse2` satisfy their corresponding contracts.

```
{-# CONTRACT reverse1, reverse2 :: {x | True}
                                -> {r | length x == length r} #-}
reverse1 x = case x of
              [] -> []
              (y:ys) -> reverse1 ys ++ [y]

reverse2 x = rev x []
{-# CONTRACT rev :: {x | True} -> {y | True}
                 -> {r | length r == length x + length y} #-}
rev x y = case x of
          [] -> y
          (a:b) -> rev b (a : y)
```

Now, consider another two popular functions `map` and `zip`. We can verify their contracts as well.

```
{-# CONTRACT map :: (Ok -> Ok) -> {xs | True}
                 -> {rs | length xs == length rs} #-}
map f xs = case xs of
            [] -> []
            (x:xs') -> f x : map f xs'

{-# CONTRACT zip :: {xs | True} -> {ys | length xs == length ys}
                 -> {rs | length rs == length xs} #-}
zip [] [] = []
zip (x:xs) (y:ys) = (x,y):zip xs ys
```

With these contracts, our system can tell that both `g1` and `g2` satisfy contract `Ok` without any unrollings.

```
g1 xs = zip xs (reverse xs)
g2 xs = zip xs (map g1 xs)
```

8.3 Sorting

As our approach gives the flexibility of asserting properties about components of a data structure, it can verify sorting algorithms. Here we give examples on list sorting. In general, our system should be able to verify sorting algorithms for other kinds of data structures, provided that appropriate contracts are given.

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

insert :: Ok -> {xs | sorted xs} -> {r | sorted r}
insert item []  = [item]
insert item (h:t) = case item <= h of
                      True -> item:h:t
                      False -> h:(insert item t)

insertsort :: Ok -> {r | sorted r}
insertsort []    = []
insertsort (h:t) = insert h (insertsort t)
```

Other sorting algorithms that can be successfully checked include `mergesort` and `bubblesort` whose definitions and corresponding annotations are shown below.

```
{-# CONTRACT fst :: (Ok, Any) -> Ok #-}
{-# CONTRACT snd :: (Any, Ok) -> Ok #-}
fst (a,b) = a
snd (a,b) = b

{-# CONTRACT merge :: {xs | sorted xs} -> {ys | sorted ys}
                   -> {r | sorted r} #-}
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys) = if x <= y
                        then x : (merge xs (y:ys))
                        else y : (merge (x:xs) ys)

split [] pair = pair
split [x] (xs, ys) = ((x:xs), ys)
split (x:y:zs) (xs, ys) = split zs ((x:xs),(y:ys))

{-# CONTRACT mergesort :: Ok -> {r | sorted r} #-}
mergesort [] = []
mergesort [x] = [x]
mergesort xs = let (as, bs) = split xs
               in merge (mergesort as) (mergesort bs)
```

```
{-# CONTRACT bsorthelper :: Ok -> {r | not (snd r) `==>`
                                       sorted (fst r)} #-}
bsorthelper :: [Integer] -> ([Integer], Bool)
bsorthelper []     = ([], False)
bsorthelper [a]    = ([a], False)
bsorthelper (x:xs) = let (y:ys, changed) = bsorthelper xs
                     in case x <= y of
                         True -> (x :(y:ys), changed)
                         False -> (y :(x:ys), True)

{-# CONTRACT bubblesort :: Ok -> {r | sorted r} #-}
bubblesort xs = let (result, changed) = bsorthelper xs
                in case changed of
                    True -> bubblesort result
                    False -> result
```

8.4 Quasi-Inference

Our checking algorithm sometimes can verify a function without programmer supplying specifications. This can be done with the help of the counter-example guided unrolling technique. While the utility of unrolling may be apparent for non-recursive functions, our technique is also useful for recursive functions. Let us examine a recursive function named `risers` [MR05] which takes a list and breaks it into sublists that are sorted. For example, `risers [1,4,2,5,6,3,7]` gives `[[1,4],[2,5,6],[3,7]]`. The key property of `risers` is that *when it takes a non-empty list, it returns a non-empty list*, which can be expressed as follows:

```
risers :: {xs | True} -> {r | not (null xs) ==> not (null r)}
```

Based on this property, the calls to both `head` and `tail` (with the non-empty list arguments) can be guaranteed not to crash. We can automatically infer this property by using counter-example guided unrolling without the need to provide a contract for the `risers` function. Consider:

```
risers [] = []
risers [x] = [[x]]
risers (x:y:etc) = let ss = risers (y : etc)
                   in case x <= y of
                       True -> (x : (head ss)) : (tail ss)
                       False -> ([x]) : ss

head (s:ss) = s
tail (s:ss) = ss
```

When a function is not annotated with a contract by programmers, we assume programmers would like to check whether the function satisfies the default contract `Ok`. We have `risers` ▷ `Ok` as follows.

```
case xs of
 [] -> []
 [x] -> [[x]]
 (x:y:etc) -> let ss = risers (y : etc)
              in case x <= y of
                  True -> (x:(head_1 ss)):(tail_1 ss)
                   False -> ([x]):ss
```

We use the label `_i` to indicate different calls to `head` and `tail`. As the pattern-matching for the parameter of `risers` is exhaustive and the recursive call will not crash, what we need to prove is that the function calls `(head_1 ss)` and `(tail_1 ss)` will not crash. Here, we only show the key part of the checking process due to space limitation. Unrolling the call `(head_1 (risers (y:etc)))` gives:

```
case (case (y:etc) of
       [] -> []
       [x'] -> [[x']]
       (x':y':etc')-> let ss' = risers (y':etc')
                      in case x' <= y' of
                          True ->(x':(head_2 ss')):(tail_2 ss')
                          False -> [x']:ss') of
 [] -> BAD
 (z:zs) -> x:z:zs
```

The branch `[]->[]` will be removed by the simplifier according to the rule *match* because `[]` does not match the pattern `(y:etc)`. For the rest of the branches, each of them returns a non-empty list. This information is sufficient for our simplifier to assert that `ss` is non-empty. Thus, the calls `(head_1 ss)` and `(tail_1 ss)` are safe from pattern-matching failure.

We refer to this simple technique as quasi-inference because the only property we infer is `Ok`. Even this simplest property `Ok` is hard to infer as the problem is undecidable in general (Section 2.2).

8.5 AVL Tree

An AVL tree is a self-balancing binary search tree and named after its two inventors, G. M. Adelson-Velsky and E. M. Landis in 1962. In Haskell, we can create a data type `AVL` as follows.

```
data Tree = L | N Int Tree Tree deriving Show
```

In an AVL tree, the depths of the two child subtrees of any node differ by at most one. The function `depth` calculates the depth of an AVL tree:

```
depth :: Tree -> Int
depth L = 0
depth (N _ t u) = (max (depth t) (depth u)) + 1
```

A function that checks whether an AVL tree is balanced is defined as:

```
balanced :: Tree -> Bool
balanced L = True
balanced (N _ t u) = balanced t && balanced u &&
                     abs (depth t - depth u) <= 1

abs :: Int -> Int
abs x = case x >= 0 of
         True -> x
         False -> -x
```

The advantage of using an AVL tree data structure is that: searching, insertion and deletion all take $O(log\ n)$ time in both the average and worst cases where n is the number of the nodes in the tree before the operation is applied. Insertion and deletion may require the tree to be rebalanced by one or more tree rotations. The idea of right rotation and left rotation is illustrated in Figure 8.1. The functions `rrotate` and `lrotate` define the two rotations respectively:

```
rrotate :: Tree -> Tree
rrotate (N v (N x l r) t) = N x l (N v r t)

lrotate :: Tree -> Tree
lrotate (N v t (N x l r)) = N x (N v t l) r
```

The function `rrotate` and `lrotate` do not require their input to be an AVL tree.

The type and a possible contract for insertion can be:

```
insert :: AVL -> Int -> AVL
{-# CONTRACT insert :: {x | balanced x} -> Ok ->
                       {r | notL r && balanced r &&
                            0 <= depth r - depth x &&
                            depth r - depth x <= 1} #-}
```

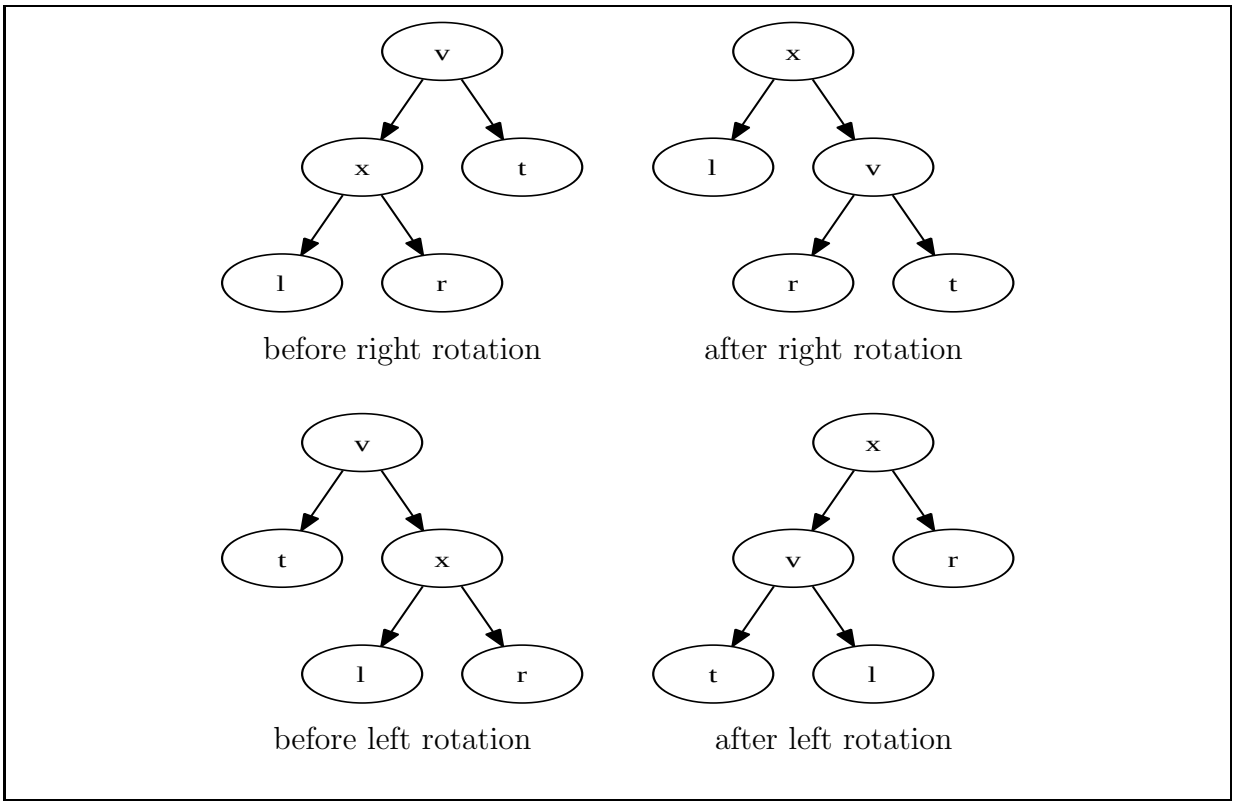

Figure 8.1: Right and left rotation

The contract of `insert` requires the input tree to be balanced. The postcondition `notL r` says that the resulting tree should not be an empty tree where `notL` is defined as:

```
notL :: AVL -> Bool
notL L = False
notL (N _ _ _) = True
```

Besides requiring the resulting tree to be balanced (indicated by `balanced r`), the post-condition also requires the depth of the resulting tree is greater than that of the input tree but the difference cannot be greater than one. It is because inserting one node to a tree may increase the depth of the tree, but due to re-balancing of the tree by proper rotations, the difference in depth should not be greater than one. The function `insert` is defined as follows.

```
insert x i
 = case x of
    L -> N i L L
    N v t u ->
     case i < v of
     True -> let t1 = insert t i
             in case depth t1 - depth u > 1 of
```

```
                True -> case t1 of
                        L -> error ‘‘insert’’
                        N x l r ->
                          case depth l > depth r of
                              True -> rrotate (N v t1 u)
                           False -> rrotate (N v (lrotate t1) u)
                False -> N v t1 u
      False -> case i > v of
               True -> let u1 = insert u i
                       in case depth u1 - depth t > 1 of
                          True -> case u1 of
                                  N x l r ->
                                   case depth l > depth r of
                                   True  -> lrotate (N v t (rrotate u1))
                                   False -> lrotate (N v t u1)
                           False -> N v t u1
               False -> N v t u
```

At the top level, there are two branches: `L` and `N v t u`. For the `L` branch, it can be easily verified that `N i L L` satisfies the postcondition of the function `insert`. For the branch `N v t u`, there are two sub-branches which are symetric. Let us consider one of them, say the `True` branch.

```
case x of
 N v t u -> -- (1)
  case i < v of
   True -> let t1 = insert t i -- (2)
           in case depth t1 - depth u > 1 of  -- (3)
               True -> case t1 of
                       L -> error ‘‘insert’’ --  (%A)
                       N x l r -> -- (4)
                        case depth l > depth r of -- (5)
                         True -> rrotate (N v t1 u)  -- (%B)
                         False -> rrotate (N v (lrotate t1) u) -- (%C)
               False -> N v t1 u  -- (%D)
```

Based on the framework described in previous chapters, we show that we can (manually) verify that the sub-branch indicated by (`%B`) produces an AVL tree. The other two branches (`%C`) and (`%D`) follow in a similar way.

First, we know the function `insert` takes an AVL tree as input (i.e. `x` is an AVL tree). That means the difference between the depth of `t` and the depth of `u` is no greater than one. From line marked by (1), by inlining `balanced (N v t u)`, and the `abs` in the definition of `balanced`, we obtain this constraint:

```
(BG_PUSH (AND (AND (balanced t) (balanced u))
              (OR (AND (>= (depth t) (depth u))
                       (AND (<= 0 (- (depth t) (depth u)))
                            (<= (- (depth t) (depth u)) 1)))
                  (AND (<= (depth t) (depth u))
                       (AND (<= 0 (- (depth u) (depth t)))
                            (<= (- (depth u) (depth t)) 1))))))
```

For the ease of reading, we write `(- (depth t) (depth u))` instead of following the definition of `abs` strictly to write `(- (- (depth u) (depth t)))`; the theorem prover does not distinguish these two representations. Here, we use Simplify's prefix notation [DNS05]. The function `depth` is treated as an uninterpreted function by the theorem prover. The key word `BG_PUSH` pushes the constraint to a truth stack. Whenever we make a query later, the prover assumes all constraints on the truth stack hold.

Second, we can make use of the fact about the postcondition of the recursive call `insert` at line marked by (2) to obtain the following constraint.

```
(BG_PUSH (AND (AND (balanced t1) (notL t1))
              (AND (<= 0 (- (depth t1) (depth t)))
                   (<= (- (depth t1) (depth t)) 1))))
```

The condition `(notL t1)` indicates that the sub-branch `L` (i.e. the branch leading to `error ``insert''`) is unreachable. Let us focus on the path leading to (`%B`). The line marked by (3) gives:

```
(BG_PUSH (> (- (depth t1) (depth u)) 1))
```

As `t1` is balanced, the line marked by (4) gives:

```
(BG_PUSH (EQ (depth t1) (+ 1 (depth l))))

(BG_PUSH (AND (AND (balanced l) (balanced r))
              (OR (AND (>= (depth l) (depth r))
                       (AND (<= 0 (- (depth l) (depth r)))
                            (<= (- (depth l) (depth r)) 1)))
                  (AND (<= (depth l) (depth r))
                       (AND (<= 0 (- (depth l) (depth r)))
                            (<= (- (depth l) (depth r)) 1))))))
```

The line marked by (5) gives directly this:

```
(BG_PUSH (> (depth l) (depth r)))
```

We have the following constraints from the call `rrotate (N v t1 u)`.

```
(BG_PUSH (OR (AND (>= (depth r) (depth u))
                  (EQ res (depth r)))
             (AND (< (depth r) (depth u))
                  (EQ res (depth u)))))

(BG_PUSH (EQ (depth u2) (+ res 1)))
```

After the right rotation, the resulting tree is `N x l (N v r u)`. First, let `res` be the maximum value of `(depth r)` and `(depth u)`. Then, let `u2` denote the depth of `(N v r u)`. We can check whether `(N v r u)` is balaned tree by sending the following query to the theorem prover:

```
(OR (AND (>= (depth r) (depth u))
         (AND (<= 0 (- (depth r) (depth u)))
              (<= (- (depth r) (depth u)) 1)))
    (AND (<= (depth r) (depth u))
         (AND (<= 0 (- (depth u) (depth r)))
              (<= (- (depth u) (depth r)) 1))))
```

If readers copy all constraints to one file and call the theorem prover Simplify, theorem prover gives the answer "Valid". Next, we want to check whether `(N x l u2)` is balanced. If so, the resulting tree at (`%B`) is an AVL tree. From above, we know `u2` is balanced; from (4), we know `l` is balanced. All we need to check is the depth difference is no greater than one:

```
       (OR (AND (>= (depth l) (depth u2))
                (AND (<= 0 (- (depth l) (depth u2)))
                     (<= (- (depth l) (depth u2)) 1)))
           (AND (<= (depth l) (depth u2))
                (AND (<= 0 (- (depth u2) (depth l)))
                     (<= (- (depth u2) (depth l)) 1))))
```

Theorem prover returns "Valid" again.

We leave `deletion` for AVL tree and `insertion` for Red-Black tree as exercise to readers.

Chapter 9

Implementation and Experiments

9.1 Embedding Static Contract Checking into GHC

We integrate the static contract checker described in this thesis to one branch of the Glasgow Haskell Compiler (GHC). The overall structure of GHC is shown in Figure 9.1. The **Verify** pass is the static contract checker, which is called after the GHC-Core is obtained. The static contract checking does not interfere with the rest of the compilation. It only generates more warning or error messages during compilation time.

9.2 Interfacing to a Theorem Prover

In this section, we give the details on how we invoke a theorem prover named *Simplify* during our symbolic simplification process. As mentioned in Section 7.1.2, we only send the scrutinee of a case-expression involving arithmetic to a theorem prover as we can handle the rest with our simplification rules. For example, when we encounter the following fragment:

```
case (i + 8 > i) of
   False -> BAD "foo"
   True -> 5
```

we would like to send the expression `i + 8 > i` to the theorem prover which will give `True` as the output. From this information, we know the fragment can be simplified to `5`. For another example:

```
... case i > j of
     True -> case j < 0 of
               False -> case i > 0 of    -- (*)
                          False -> BAD
     False -> case j > 5 of
                True -> ...
```

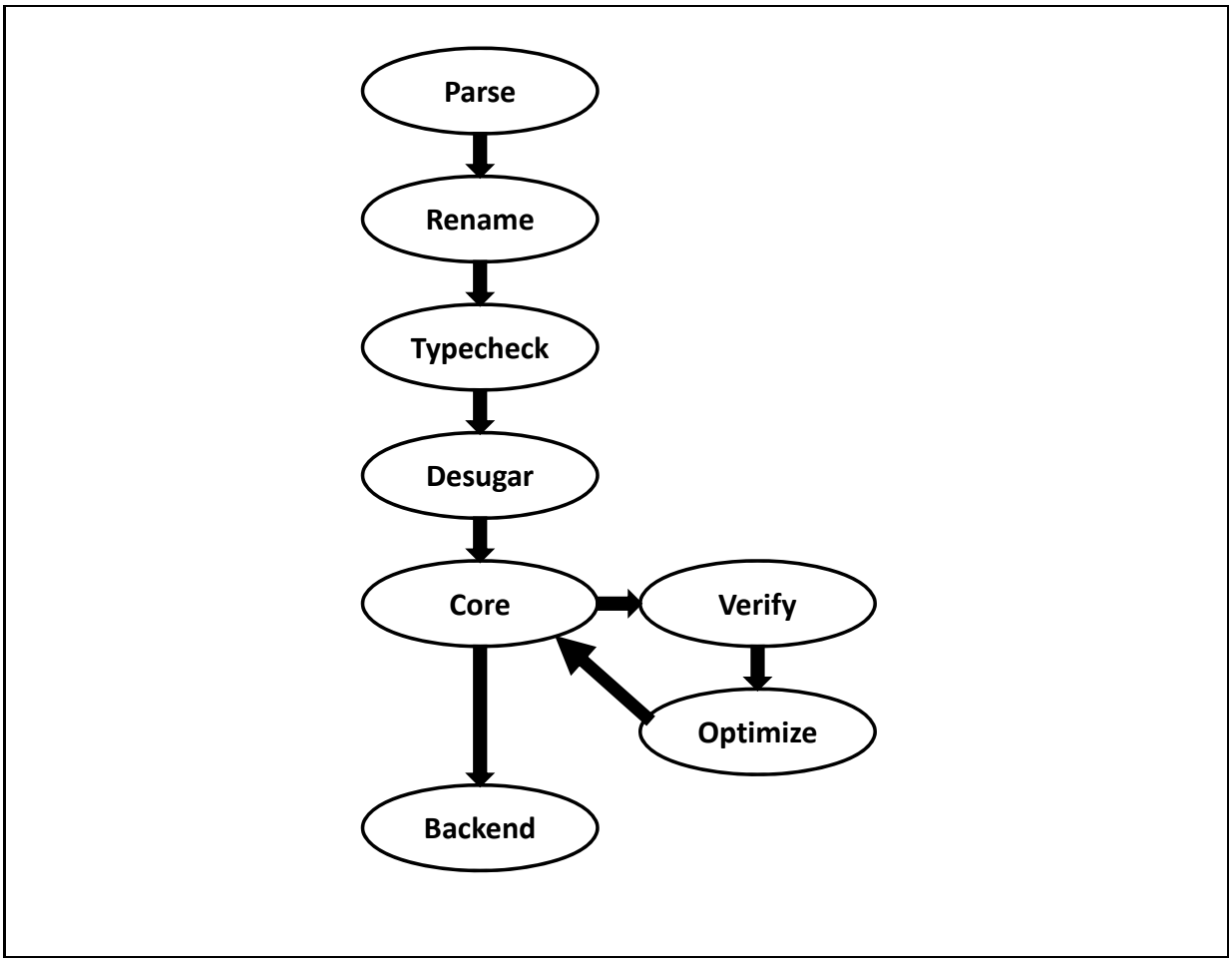

Figure 9.1: Overall Structure of GHC

In this case, when we reach the line marked by `(*)`, before we query `i > 0`, we should send information `i > j == True` and `j < 0 == False` to the theorem prover. Fortunately, the theorem prover *Simplify* provides a stack for us to push known information to the stack and pop it out when it is no longer valid or needed. In the above example, we push `i > j == True` and `j < 0 == False` to the stack before query `i > 0`; when we work on the `False` branch of `i > j`, we know the information `j < 0 == False` is no longer valid so we pop it out from the stack before we query `j > 5`. The pushing and popping can be done with the command `BG_PUSH` and `BG_POP` provided by the theorem prover *Simplify*.

Due to our special need mentioned above, we would like to call the theorem prover interactively by sending pieces of information one by one and making small queries one after another. Fortunately, in GHC's base library, there is a module `System.Process` which contains a function named `runInteractiveCommand` that allows us to run a command using the shell (e.g. invoking the theorem prover *Simplify* by a command `Simplify.exe`), and return `Handle`s that may be used to communicate with the process via its `stdin`, `stdout`, and `stderr` interactively. This means we need to do the following steps:

1. Start the theorem prover. The function `startProver` is invoked before the start of the ESC/Haskell algorithm. Basically, we create two `Handle`s: `inH` and `outH`, to handle input and output respectively after invoking `Simplify.exe` using the shell.

```
data Prover = P { inH, outH :: Handle }

startProver :: IO Prover
startProver
 = do { (pin,pout,perr,pid) <- runInteractiveCommand "Simplify.exe"
      ; hSetBuffering pin LineBuffering
      ; let p = P { inH = pin, outH = pout }
      ; return p}
```

2. Convert an expression involving arithmetic to `String` form, which is in the acceptable *formula* format, before sending it to the theorem prover.

```
send :: Prover -> String -> IO ()
send p s = hPutStrLn (inH p) s

pushProver :: Prover -> Exp -> IO ()
pushProver p e
 = send p (formulaToString (BGPush [toFormula e]))

popProver :: Prover -> IO ()
popProver p = send p (formulaToString BGPop)
```

3. Once we get back the result (of type `String`) from the theorem prover, we convert it back to an expression by parsing the `String`. We read one line at a time until we find either the word "Valid" or the word "Invalid", which can be successfully parsed by the parser.

```
findResponse :: Show a => Prover->Parser a->IO a
findResponse prover parser
 = do { -- Reading from Simplify
      ; line <- hGetLine (outH prover)
      ; case parse parser "" line of
       Right b  -> return b -- Got result
       Left err -> findResponse prover parser
      }
```

4. Stop the theorem prover. The function `stopProver` is invoked after the end of the ESC/Haskell algorithm.

```
stopProver :: Prover -> IO ()
stopProver (P { inH = pin, outH = pout })
 = do { hClose pin; hClose pout }
```

The theorem prover *Simplify* gives two possible outputs: `Valid` and `Invalid`. If the result is `Invalid`, a counter-example is given as well. For example, the result of a query `i + 8 > i` is `Valid`. However, if you query `i > j`, the result is `Invalid` and a counter-example `i <= j` is given. Similarly, if you query `not (i > j)`, the result is `Invalid` and

a counter-example `i > j` is given. This means when the theorem prover says `Invalid`, we cannot assume the negation of the formula is `Valid`. Thus, we apply the following strategy:

1. if the answer is `Valid`, take it as the result;
2. if the answer is `Invalid`, check the negation of the formula;
3. if the negation is `Valid`, take its negation as the result;
4. if the negation is `Invalid`, it means `DontKnow`.

9.3 Contract Positions

Is the following program erroneous?

```
{-# CONTRACT Pos = {x | x > 0} #-}
{-# CONTRACT h1 :: Pos -> Pos #-}
h1 x = x

g1 x = ... h1 (-1)...
```

The contract for `h1` is too strong. By too strong, we mean the programmer specified precondition is too strong. This makes the system reject programs that are actually safe. That means, in the above case, we give a false alarm.

Is below an erroneous program?

```
{-# CONTRACT h2 :: Ok -> {r | r > -7} #-}
h2 x = x*x

g2 x = ... (if h2 5 >=0 then True else BAD) ...
```

The contract for `h2` is too weak. By too weak, we mean the postcondition is too weak. It does not give the system enough information to accept those safe programs. Again, we give false alarm.

In the above two cases, if we inline the function body during CEG unrolling, we can reduce false alarms. That means at call sites `(h1 -1)` and `(h2 5)`, we inline `h1` and `h2`, and the information obtained from the function bodies can tell us that both programs are actually safe. However, by doing this, we may violate programmers wish. For example, the programmer would like a function `h1` to have contract `Pos -> Pos`, but during implementation, she makes a mistake, or she may like to change her implementation in future. What she wants other parties to know is the interface for `h1`, i.e. the contract of `h1` so that even if she changes the implementation in future, the callers of `h1` do not need to amend their code. So shall we respect the function definition more or the contracts more? We give a detailed discussion below.

```
{-# CONTRACT len :: Ok -> Pos #-}
len :: [a] -> Int
len [] = 0
len (x:xs) = 1 + len xs

f x = case len [] == 0 of
       True  -> x
       False -> BAD

g x = case len x >= 0 of
       True  -> True
       False -> BAD
```

Strong Contract Position – Respect Contract More If programmers specify $f \in t$, then we never inline f. CEG unrolling only applies to a function that does not have a contract.

Under this position, if `len` is given a contract, we report that "`f` may crash, `g` is ok". As the only information we have is postcondition of `len` is greater then 0, `len [] == 0` gives `False` while `len x >= 0` gives `True`. On the other hand, if `len` is not given a contract, we report that "`f` is ok, `g` may crash". After we inline the definition of `len`, we know that `len [] == 0` gives `True`. As we cannot prove `len x >= 0` after a number of unrollings, we give up and report that "`g` may crash".

Weak Contract Position – Respect Definition More Even if programmers specify $f \in t$, we still use CEG unrolling for f provided $f \triangleright t$ is crash-free. That means as long as f satisfies its contract though the contract may be too strong or too weak, we can always inline f's definition.

Strong-Weak Contract Position – Strong Contravariant, Weak Co-variant Another alternative could be that programmers can give stronger precondition and weaker postcondition. These conditions may result in imprecise analysis so that more programs may be rejected. However, we still choose to use CEG unrolling to inline function's body regardless it is annotated with contracts or not. After unrolling, as more precise information is obtained, the analysis result may be different from the result without unrolling. For example, in the definition of `g2`, if we do not unroll the call `(f2 5)`, the postcondition `(f2 5) >= -7` does not imply `(f2 5) >= 0`. On the contrary, if we unroll the call `(f2 5)`, we will get `(5*5)>=0` which is True, thus, we know the call to `f2` is safe. So this approach is not desirable.

Strong Flag We propose to use the strong contract position and weak contract position approaches, but not the strong-weak contract position approach. It is possible to allow programmers to flag strong contract position by using a key word `STRONG`, for example:

```
{-# STRONG CONTRACT h1 :: Pos -> Pos #-}
{-# CONTRACT h2 :: Ok -> {r | r > -7} #-}
```

This means the programmer would like us to respect the strong contract as she may change the implementation of `h1`, all the callers should assume the strong contract at call sites. On the other hand, the default contract annotation says that, respect the contract in general, but when the contract is too weak, inlining the body of `h2` to obtain more precise information is allowed.

9.4 Experiments

The experiments have been done on a PC with dual processor of speed 2.7GHz and 2GB memory running windows vista. We evaluate our tool on a medium-sized program, which consists of some small examples used in this thesis and some small programs for basic testings. The total number of lines of code is 325. The number of lines of contract annotation is 22. The time for verification is 1.17sec. This brief figure is to show that the contract checking framework can work efficiently in practice. We leave sophisticated experiments to future work. Programs under testing are shown in Appendix C.

We have manually proved the correctness of the following motivating algorithms based on the technique introduced in this thesis. We could be the first to prove properties about AVL trees.

- **List Operations** It contains the following functions that manipulate lists: `length`, `append`, `reverse`, `map`, `filter`, `take`, `drop`, `zip`, `last`, `risers`. Contracts, that are checked for these functions, are shown below.

```
{-# CONTRACT length :: {x | True} -> {r | r >= 0} #-}
{-# CONTRACT append :: {xs | True} -> {ys | True} ->
                    {r | (length xs + length ys)  == (length rs)} #-}
{-# CONTRACT reverse :: {x | True} -> {r | sameLen x r} #-}
{-# CONTRACT map :: {f | True} -> {xs | True}
                 -> {r | sameLen xs r} #-}
{-# CONTRACT filter :: {f | True} -> {x | True} -> {r | all f r} #-}
{-# CONTRACT take :: {n | n >= 0} -> {xs | True}
                  -> {r | len r <= len xs} #-}
{-# CONTRACT drop :: {n | n >= 0} -> {xs | (len xs) >= n}
                  -> {r | (len r) <= (len xs)} #-}
{-# CONTRACT zip :: {xs | True} -> {ys | sameL xs ys}
                 -> {r | sameLen xs r} #-}
{-# CONTRACT last :: {ys | not (null ys)} -> {r | True} #-}
{-# CONTRACT risers :: {x | True} -> {r | True} #-}
```

- **Sorting Algorithms** It contains these sorting algorithms: insertion-sort, merge-sort, bubble-sort and quick-sort. We first prove the `sorted` property (Section 8.3). To prove the *permutation* property, we need to have one more simplification rule:

$$\begin{array}{lll} & f\ (\texttt{case}\ e_0\ \texttt{of}\ \{pt_i \to e_i\}) & (\textsc{AppCase}) \\ \Longrightarrow & \texttt{case}\ e_0\ \texttt{of}\ \{pt_i \to (f\ e_i)\} & \text{where } f \text{ is strict} \end{array}$$

The job of the rule AppCase is to push the function f to the branches of a case-expression. This rule only holds when the function f is strict. As there is a strictness analyser readily built in GHC, we can use it to get the strictness information of the function before the phase of verification.

- **AVL Tree** See Section 8.5.
- **Red-Black Tree** See Section 8.5 and Appendix B.

Part III

Chapter 10

Possible Enhancements

10.1 Conjunctive Contracts and Disjunctive Contracts

The first extension is to allow programmers to declare multiple contracts because *some properties can not be expressed precisely with a single contract.* For example:

```
{-# CONTRACT g :: {x | x > 0} -> Any -> Ok #-}
{-# CONTRACT g :: {x | x <= 0} -> Ok -> Ok #-}
g x y = if x > 0 then x + 1
                 else y
```

One might think to combine the two contracts into one:

```
g :: x:Ok -> {y | x <= 0} -> Ok
```

But this is a stronger contract which is actually equivalent to the following multi-contracts:

```
{-# CONTRACT g :: {x | x > 0} -> {y | False} -> Ok #-}
{-# CONTRACT g :: {x | x <= 0} -> Ok -> Ok #-}
```

We know that `{y | False}` and `Any` are not the same by the definition in Figure 4.3 in Section 4.3. In order to express more precise properties, we allow programmers to give multiple contracts to a function.

We propose to add both *conjunctive contracts* and *disjunctive contracts*:

$$
\begin{array}{lcll}
t & \in & \textbf{Contract} & \\
t & ::= & \ldots & \\
 & | & t_1 \wedge t_2 & \text{Conjunctive Contract} \\
 & | & t_1 \vee t_2 & \text{Disjunctive Contract}
\end{array}
$$

The definition of contract satisfaction could be:

$$
\begin{array}{lcl}
e \in t_1 \wedge t_2 & \iff & e \in t_1 \text{ and } e \in t_2 \\
e \in t_1 \vee t_2 & \iff & e \in t_1 \text{ or } e \in t_2
\end{array}
$$

The definition of $\triangleright$ could be:

$$\begin{array}{lcl} e \triangleright t_1 \wedge t_2 & \Longleftrightarrow & e \triangleright t_1 \triangleright t_2 \\ e \triangleright t_1 \vee t_2 & \Longleftrightarrow & (\!|e \triangleright t_1, e \triangleright t_2|\!) \end{array}$$

For disjunctive contracts, we may have to introduce a special constructor $(\!|, |\!)$ as we would like to keep both results: $e \triangleright t_1$ and $e \triangleright t_2$. But it is not easy to define such a constructor that fits our algorithm of contract wrappers. A lot of research has to be done.

In [HJL06], Hinze et. al. mention that conjunctive contracts and disjunctive contracts are not necessarily projections. Thus, we need to check whether the following holds:

$$\begin{array}{lll} \text{Idempotency:} & (a)\ e \triangleright (t_1 \wedge t_2) \triangleright (t_1 \wedge t_2) \equiv e \triangleright (t_1 \wedge t_2) \\ & (b)\ e \triangleleft (t_1 \wedge t_2) \triangleleft (t_1 \wedge t_2) \equiv e \triangleleft (t_1 \wedge t_2) \\ & (c)\ e \triangleright (t_1 \vee t_2) \triangleright (t_1 \vee t_2) \equiv e \triangleright (t_1 \vee t_2) \\ & (d)\ e \triangleleft (t_1 \vee t_2) \triangleleft (t_1 \vee t_2) \equiv e \triangleleft (t_1 \vee t_2) \\ \text{Approximate Identity:} & e \in t_1 \wedge t_2 \Rightarrow \ (a)\ e \preceq e \triangleright t_1 \wedge t_2 \\ & \qquad\qquad\qquad\ \ (b)\ e \triangleleft t_1 \wedge t_2 \preceq e \\ & e \in t_1 \vee t_2 \Rightarrow \ (c)\ e \preceq e \triangleright t_1 \vee t_2 \\ & \qquad\qquad\qquad\ \ (d)\ e \triangleleft t_1 \vee t_2 \preceq e \end{array}$$

10.2 Recursive Contracts

Only with the concept of disjunctive contracts, we can define recursive contracts. With recursive contracts, we can *specify properties for a recursively defined data structure without defining a recursive predicate*. Without recursive contracts, we cannot give a contract to a list of elements where some of the element may be undefined.

Below is not an ideal design, because it only works with polymorphic data types.

```
length :: [Any] -> Ok
length [] = 0
length (x:xs) = 1 + length xs
```

where the shorthand `[Any]` is actually the `AnyList` below.

```
data List a = Cons a (List a) | Nil

{-# CONTRACT AnyList = List Any #-}
```

We may want something like

```
{-# NEWCONTRACT AnyList = Cons Any AnyList | Nil #-}
```

10.3 Polymorphic Contracts

The forms of contracts seem very rich already, however, with these we still cannot give a precise contract for the popular function `map`. We need quantifiers! It is known to be more complex when quantifiers are involved. But they are very useful especially together with recursive contracts. Here, we propose to introduce *polymorphic contracts* in which we quantify *contract variables* instead of quantifying over program variables or type variables.

We may also add a rank-2 contract, which is more expressive than a conjunctive contract because the former allows arbitrary instantiation of a contract variable while the later one explicitly specifies the instances expected. These extensions greatly increase the expressive power of contracts so that we can verify more precise properties for higher-order functions.

t	$\in$	**Contracts**	
t	$::=$	$\ldots$	
	$\mid$	β	Contract Variable
	$\mid$	$\forall\beta.t$	Quantified Contract

If we have contract variable, we can define the contract of `map` as follows:

```
{-# CONTRACT map :: forall t1, t2. (t1 -> t2) -> [t1] -> [t2] #-}
```

where `t1` and `t2` are contract variables. For example, `map head` may have contract:

```
[NonNull] -> [Ok]
```

As the polymorphic type of `map` forces the function `f`'s parameter and the second parameter of `map` to have the same contract. For example, if we have `map head [[1],[],[2]]`, we can see that the expression `[[1],[],[2]]` does not satisfy the contract `[NonNull]`. Thus, we know that the call `map head [[1],[],[2]]` will crash.

The contract of `map` looks exactly the same as its polymorphic type:

```
map :: forall a, b. (a -> b) -> [a] -> [b]
```

where `a` and `b` are type variables. Although the type and contract of `map` look the same, one cannot replace another. Consider:

```
mapInt :: (Int -> Int) -> [Int] -> [Int]
```

where `mapInt` is a monomorphic version of `map` that works on integers only. We can give it the same contract as `map`, i.e.

```
{-# CONTRACT map :: forall t1, t2. (t1 -> t2) -> [t1] -> [t2] #-}
```

With this contract, a call `(map pos)` may have contract `[Pos] -> [Pos]` assuming the contract of `pos` is `Pos` which is `{x | x > 0}`. This definitely cannot be achieved by the type declaration of `mapInt`.

On the other hand, polymorphic contracts may cause more false alarms. Consider:

```
{-# CONTRACT twice :: forall t. (t -> t) -> t -> t #-}
twice :: forall a. (a -> a) -> a -> a
twice f x = f (f x)
```

Programmers give the function `twice` an appropriate type, but a contract that is too strong. As `f` is only applied twice, it is not necessary to enforce the function to have the same precondition and postcondition. For example, we have:

```
tail :: NonNull -> Ok
tail (x:xs) = xs
```

A call to `(twice tail [1,2])` is safe. As the precondition and the postcondition of `tail` are not the same, we may give a false alarm in this case. We believe that this is an interesting area to explore.

10.4 Declaration of Lemmas

A contract is for programmers to specify the input-output relation of *one* function. It is not for specifying a meta property of a function, for example, the associativity of (+). It is not for specifying a relation between functions, either. For example, we cannot specify `pop (push s a) == a` where the function `push` pushes an item to the top of a stack and the function `pop` pops out the topmost element from the stack. Thus, in addition to contracts, it is ideal to allow programmers to declare some simple *lemmas* which may help a lot in assisting the verification process. For example, we need `lemma1` to prove the postcondition of the function `selectMin` which is used in the definition of `selectionSort`.

```
lemma1 x y xs
  = smaller x xs && y <=x => smaller y xs

smaller x [] = True
smaller x (y:ys) = x <= y && smaller x ys

{-# CONTRACT selectMin :: {xs | not (null xs)} ->
                          {rs | smaller (fst rs) xs } #-}

selectMin [x]     = (x, [])
selectMin (x:xs') = let y  = fst (selectMin xs)
                        ys = snd (selectMin xs)
                    in if x <= y then (x, xs')
                                 else (y, (x:ys))

{-# CONTRACT selectionSort :: Ok -> {r | sorted r} #-}
selectionSort []     = []
selectionSort (x:xs) = let (y,ys) = selectMin (x:xs)
                       in y : selectionSort ys
```

Other useful lemmas include:

```
lemma2 xs ys
  = length (xs ++ ys) == length xs + length ys
lemma3 xs
  = reverse (xs ++ ys) == reverse ys ++ reverse xs
```

These lemmas need to be verified before being applied. As lemmas are just functions, we can use our existing system to verify them. We can give each lemma a contract specifying that its result must be `True`. For example:

```
{-# CONTRACT lemma1 :: Ok -> Ok -> Ok -> True #-}
{-# CONTRACT lemma2 :: Ok -> Ok -> True #-}
{-# CONTRACT lemma3 :: Ok -> True #-}
```

If a lemma satisfies its contract, the lemma is valid. As the contracts of these lemmas share a common pattern (i.e. the contract for each parameter is `Ok`, the contract for the result is `True`), these contracts can be automatically generated. QuickCheck [CH03] is an automatic random testing tool for Haskell programs. It is made up of a combinator library written in Haskell which contains many such lemmas. Verifying the lemmas in their library could be a good case study and also can test the verification strength of our system.

We have shown how lemmas can be defined and verified, now we explore how lemmas can be applied. We may introduce a new key word `using` so that programmers can write something like: e `using` $\texttt{lemma}_i\ \vec{a}$ for some i. This expression is then transformed to

$$\begin{array}{l}\texttt{case}\ \texttt{lemma}_1\ \vec{a}\ \texttt{of}\\ \quad \texttt{True} \rightarrow e\\ \quad \texttt{False} \rightarrow \texttt{UNR}\end{array}$$

Another alternative is to make use of the external theorem prover by pushing all lemmas to the *truth* stack. These are just our first thoughts. You might notice that `lemma3` does not hold if `xs` is infinite and `ys` is finite. , we may adapt some ideas from [FPST07], which supports both dependent types and lemmas.

10.5 Data Type Invariants

Besides giving contracts to functions, programmers may like to supply contracts for data constructors as well. For example:

```
data S a where
 S1 :: Int -> Int -> S Int
 {-# CONTRACT S1 :: {x | True} -> {y | x > y} -> {r | True} #-}
 S2 :: S Int -> S Int
 {-# CONTRACT S2 :: S {x | True} -> S {r | r > x} #-}
```

It says that when `S1` is constructed, it should take two integers where the first one is greater than the second one. For example, `(S1 5 4)` is a valid expression while `(S1 1 2)` is not. When `S1` is pattern-matched, the invariant `(x > y)` can be assumed, e.g.:

```
{-# CONTRACT f :: S Int -> Int #-}
f (S1 x y) = 1/(x - y)
```

We know that the call to the division function `(/)` will never fail because `x-y` is always greater than `0`. The example `S2` shows how similar things can be done with GADT [XCC03, JVWW06].

Note that this is totally different from the constructor contracts mentioned in Section 4. Constructor contracts are about constructing contracts using data constructors while this data type invariant is about giving contracts to data constructors.

In order to cater for invariants of data types, we need to pay attention to two places in a program:

(1) the return of the data constructors.

(2) the pattern-matching of the data constructors.

For the case (1), we can treat a data constructor as a normal function and the invariant can be viewed as a precondition to the constructor. For example:

```
d1 x = S1 (x + 1) x
```

The checking code for `d1` is:

$$\begin{array}{rcl} \texttt{d1} \triangleright \texttt{Ok} \to \texttt{Ok} & = & (\lambda x.\ ((\texttt{S1} \triangleleft x : \texttt{Ok} \to \{y \mid x > y\} \to \texttt{Ok})\ (x+1)\ x)) \triangleright \texttt{Ok} \to \texttt{Ok} \\ & = & \ldots \\ & = & \lambda x.\ \texttt{S1}\ (x+1) \left(\begin{array}{l} \texttt{case}\ (x+1) > x\ \texttt{of} \\ \quad \texttt{True} \to x \\ \quad \texttt{False} \to \texttt{BAD} \end{array} \right) \end{array}$$

As $(x + 1) > x$ holds, the `BAD` is unreachable. So we know that `d1` satisfies the contract $\texttt{Ok} \to \texttt{Ok}$.

For the case (2), we treat the invariant as a postcondition and this information can be very useful after a pattern is matched. Recall the example `f`:

```
f s = case s of
        (S1 x y) -> 1 / (x - y)
```

Suppose the contract for the division operator $(/)$ is $\texttt{Ok} \to \{q \mid q \neq 0\} \to \texttt{Ok}$. The checking code for `f` is as follows.

$$
\begin{array}{lcl}
\texttt{f} \triangleright \texttt{Ok} \to \texttt{Ok} & = & \lambda s.\ \texttt{case}\ s\ \texttt{of} \\
& & \quad (\texttt{S1}\ x\ y) \to\ \texttt{case}\ (\texttt{S1} \triangleright x : \texttt{Ok} \to \{y \mid x > y\} \to \texttt{Ok})\ x\ y\ \texttt{of} \\
& & \qquad \texttt{True} \to ((/) \triangleleft \texttt{Ok} \to \{q \mid q \neq 0\} \to \texttt{Ok})\ 1\ (x - y) \\
& = & \ldots \\
& = & \lambda s..\ \texttt{case}\ s\ \texttt{of} \\
& & \quad (\texttt{S1}\ x\ y) \to\ \texttt{case}\ x > y\ \texttt{of} \\
& & \qquad \texttt{True} \to\ \texttt{case}\ (x - y) \neq 0\ \texttt{of} \\
& & \qquad\quad \texttt{True} \to 1/(x - y) \\
& & \qquad\quad \texttt{False} \to \texttt{BAD}
\end{array}
$$

As $(x > y)$ implies $(x - y) \neq 0$, we know the `BAD` is unreachable. From this example, we can see that in the checking code of `f`, we need also to add an extra case-expression for each pattern matching. That means for each function call f_i in the body of f, it is not enough to simply replace the function call f_i by $f_i \triangleleft t_i$ where t_i is the contract of f_i. We define an operator $[\![.]\!]^{\#}$ (in Figure 10.1) that does two things: (1) replace f_i by $f_i \triangleleft t_i$; (2) add extra case-expression for matched patterns. The function ρ takes either a variable for a data constructor as input and fetches its contract.

$$
\begin{array}{lcl}
[\![.]\!]^{\#} :: \mathbf{Exp} \to \mathbf{Exp} & & \\
[\![n]\!]^{\#} & = & n \\
[\![v]\!]^{\#} & = & v \triangleleft \rho(v) \\
[\![\lambda x.e]\!]^{\#} & = & \lambda x.[\![e]\!]^{\#} \\
[\![K\ \vec{e}]\!]^{\#} & = & \rho(K)\ \vec{[\![e]\!]^{\#}} \\
[\![e_1\ e_2]\!]^{\#} & = & [\![e_1]\!]^{\#}\ [\![e_2]\!]^{\#} \\
[\![\texttt{let}\ x = e_1\ \texttt{in}\ e_2]\!]^{\#} & = & \texttt{let}\ x = [\![e_1]\!]^{\#}\ \texttt{in}\ [\![e_2]\!]^{\#} \\
[\![\texttt{case}\ e_0\ \texttt{of}\ \{C_i\ \vec{x_i} \to e_i\}]\!]^{\#} & = & \texttt{case}[\![e_0]\!]^{\#}\ \texttt{of} \\
& & \quad \{K_i\ \vec{x_i} \to \texttt{case}\ (K_i \triangleleft \rho(K_i))\ \vec{x_i}\ \texttt{of} \\
& & \qquad\quad \texttt{True} \to [\![e_i]\!]^{\#}\}
\end{array}
$$

Figure 10.1: Abstraction Derivation

10.6 Lazy Dynamic Contract Checking

For those contracts that we fail to check during compile time (i.e. we do not know $e \in t$ or not), we would like to check the contract during run-time. For example, we may define a function `assert` that checks whether the predicate `b` holds at run-time:

```
assert :: True -> a -> a
assert b x = case b of
               True -> x
               False -> error ``Assertion failed!''
```

As our contract checking technique adopts the idea from dynamic contract checking [FF02, HJL06], we can use the same ▷ algorithm to check those contracts, which we fail to check statically, at run-time. Recall the function `insert` in Section 8.3.

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

insert :: Ok -> {xs | sorted xs} -> {r | sorted r}
insert item []  = [item]
insert item (h:t) = case item <= h of
                      True -> item:h:t
                      False -> h:(insert item t)
```

In case we cannot statically verify that `insert` satisfies its contract, we may generate the following dynamic checking code to check the contract:

```
checkedInsert x xs = let vs = (assert (sorted xs) xs))
                     in assert (sorted (insert x vs)) (insert x vs)
```

To make our presentation simple, let us focus on precondition checking only:

```
checkedPreInsert x xs = insert x (assert (sorted xs) xs)
```

It works fine in a strict language, for example:

```
> checkedPreInsert 3 [1,5,2,4]
Assertion failed!
```

However, in a lazy language, it fails to work as the assertion may not terminate. For example:

```
take 5 (checkedPreInsert 3 [1,2..])
```

goes into an infinite loop while the original unchecked code

```
take 5 (insert 3 [1,2..])
```

gives `[1,2,3,3,4]`.

It is non-trivial to insert an assertion as lazy as possible. Existing work on this problem can be divided into two strands:

- *strict dynamic contract checking* [FF02, HJL06].
- *lazy assertion* [CMR03, CH06, CH07].

Obviously, strict dynamic contract checking is not the most appropriate technique for a lazy programming language. Furthermore, lazy assertions cannot assign blame in the same precise way as contracts. Findler et al. [FyGR07] took a step towards lazy dynamic contract checking by adding "a small controlled amount of laziness" to contract checking, but this is still a fertile research area. Lazy dynamic contract checking will help in the debugging phase of software development. All run-time checks related to contracts can be switched off when building the final version of the software.

10.7 Program Optimization

Contracts are not only useful for verifying program's safety, they may also help in optimizing the program, in particular, *dead code elimination* and *redundant code elimination*.

10.7.1 Dead Code Elimination

Dead code refers to a piece of code that is not reachable during the execution of a program. Here, we propose to eliminate dead pattern matchings. For example:

```
zip :: [a] -> [b] -> [(a,b)]
{-# CONTRACT zip :: xs:Ok -> {ys | length xs == length ys} -> Ok #-}
zip [] []         = []
zip (x:xs) (y:ys) = (x,y): zip xs ys
```

If we know the function `zip` which always takes two lists of the same length, with its contract, we do not need to pattern match its second argument. Xi [Xi99] has addressed this kind of dead pattern elimination with dependent-type checking and mentioned that this can contribute up to 20% speedup. As our contracts are much more expressive than the restricted version of dependent type in [Xi99], we have potential to eliminate more dead patterns than theirs.

10.7.2 Redundant Array Bound Checks Elimination

Redundant code can be reached during the execution of a program but does not have any effect on the subsequent code to be executed. People have worked on eliminating redundant array bound checks for more than two decades. We hope the proposed contract framework is an injection of new blood to this area. The target application is software related to graphics, animation and finance which involve tremendous array accesses.

A simple example of array bound elimination is illustrated below. Assume the function `sub` performs direct array access and has the following type:

```
sub :: Array Int b -> Int -> b
```

Another function `(!)`, which performs bounds checking before accessing the element of an array, is defined as follows.

```
{-# CONTRACT (!) :: {a | True} -> {i | i>=0 && i < bound a} -> Ok #-}
(!) arr i = if i >= 0
              then if i < bound arr then sub arr i
                    else error "Index is too large!"
              else error "Index is too small!"
```

Consider a function `sumArr` which sums up all the elements in an array:

```
sumArr :: Array Int Int -> Int
sumArr arr = sumArrAux arr (bound arr - 1)

sumArrAux arr n = case n of
                    0 -> arr!n
                    _ -> arr!n + sumArrAux arr (n - 1)
```

Our system may infer that `sumArr` satisfies contract `Ok`, which means the precondition of `(!)` is always satisfied. This implies that we can replace `(!)` by `sub` (i.e. removing the two bound checks) and improve the efficiency of running `sumArr` greatly.

10.8 Detecting Divergence or Termination

Consider:

```
d1 = case (length [1..] == length [2..]) of
       True -> 1
       False -> BAD
```

After inlining both calls to `length` for a fixed number of times, we stop and say we are not able to tell the two calls are equal. That means we cannot prove `True` is a sufficient precondition of `d1`. The program diverges when evaluating `length [1..] == length [2..]`, however, we cannot tell. Thus, in this case, we will raise a false alarm.

We can detect some trivial divergence. For example:

```
bot :: a -> a
bot x = bot x

d2 = case bot True of
       True -> 1
       False -> BAD
```

Syntactically, we can detect that the function `bot` diverges no matter what argument it takes simply because the recursive parameter is the same as the formal parameter. In this case, we know that `bot True` will diverge, so the branch leading to `error` is unreachable.

On the other hand, we may try to detect termination. There are a few works on termination analysis to functional programs [LJBA01, SJ05, Ser07], but not for a lazy language, and not modular in the presence of higher-order functions. One automatic termination analysis for Haskell programs that makes uses term rewriting technique is discussed in [GSSKT06]. Their analysis is modular in the presence of higher-order functions, but it is implemented in Java, which makes full integration into GHC challenging.

Chapter 11

Related work

Static verification of software is a field dense with related work, of which we can only summarise a limited fraction here.

11.1 Contracts

The idea of "contract" was first established by Parnas [Par72] and popularized by Meyer in its use in Eiffel [Mey92]. More recently, Findler and Felleisen introduced the notion of higher-order contracts, including a careful treatment of "blame" [FF02]. This paper unleashed a new wave of papers about contract checking in higher order languages, including [BM06, FB06, BM06, HJL06, WF07, Fla06, KTG+06, KF07, GF07]. Although they share a common foundation, these papers differ in their notation and approach, which makes like-for-like comparisons difficult.

Of these papers, the work of Blume and McAllester [BM06, BM06] is by far the most closely related because they give a declarative semantics for contract satisfaction, and prove a connection with the dynamic wrappers of Findler and Felleisen. Here is a brief summary of the differences between some of this work, especially [BM06], and our own:

- We aim at static contract checking, for a statically typed language, whereas most of the related work deals with dynamic checks, or a hybrid checking strategy for a dynamically typed language.
- We deal with a lazy language; all other related work is for strict languages. In particular, we give a crashing expression a contract `Any` while a contract is only given to non-crashing expressions in [FF02, BM06, FB06].
- We deal with dependent function contracts which [FF02, FB06] do not.
- We lay great emphasis on crashing and diverging contracts, which are either not the focus of these other works, or are explicitly excluded. Our solution appears both less restrictive than [BM06], by allowing crashing functions to be called within contracts, and supports Theorem 9 in both directions, rather than the ($\Leftarrow$) direction only.

- Our definition of contract satisfaction (i.e. the denotational semantics for contracts) is different from Blume & McAllester's [BM06]. Besides the extra contract `Any` introduced, another difference is that we require expressions that satisfy a predicate contract to be *crash-free* while they do not have this requirement. (The reason for having this requirement is discussed in Section 5.1.2.) As a result, all our contracts are inhabited by some expressions while the contract $\{x \mid \texttt{False}\}$ is not inhabited by anything in [BM06]. (The reason why it is good to have all contracts inhabited is stated in Section 4.3.4.)

- The telescoping property (Figure 5.2) is first discovered in [BM06], but it does not seem to be used in any of the proofs in [BM06] while we use it intensively to make many proofs of our lemmas much simpler.

- Findler and Blume discovered that contracts are pairs of projections in [FBF06, FB06]. That means given a contract t, $\lambda e.\mathcal{W}_t(e)$ is a projection where $\mathcal{W}_t$ is a wrapper function. To be a projection w.r.t. $\sqsubseteq$, a function p must satisfy these two properties:

  1. $p \circ p = p$ (idempotence)
  2. $p \sqsubseteq 1$ (result of projection contains no more information than its input)

  Our $(\bullet \triangleright t)$ and $(\bullet \triangleleft t)$ (i.e. $\lambda x.(x \triangleright t)$ and $\lambda x.(x \triangleleft t)$) satisfy the idempotence property as shown in Figure 5.2, but does not satisfy (2). They only satisfy (2) under the condition that the input of the projection satisfies its contract t as shown in Figure 5.2. Moreover, we discover the *projection pair* property (in Figure 5.2), which plays a crucial role in our proof.

- Blume & McAllester deal with recursive contracts, which we do not.

Inspired by [FF02, BM06], Hinze et al [HJL06] implement contracts as a library in Haskell and contracts are checked at run-time. The framework also supports contract constructors such as pairs, lists, etc. Another dynamic contract checking work is the Camila project [VOS+05] which use monads to encapsulate the pre/post-conditions checking behaviour.

The hybrid contract checking framework [Fla06, KTG+06, KF07, GF07], in theory, can be as powerful as our system. (Hybrid checking means a combination of static and dynamic contract checking.) But in practice, our symbolic execution strategy adopted from [Xu06] gives more flexibility to the verification as illustrated in §2.1.3. In [WF07], Wadler and Findler show how contracts fit with hybrid types and gradual types by requiring casts in the source code. The casts are similar to the job of our $\triangleright$ and $\triangleleft$.

11.2 Verification Condition Generation Approach

In an inspiring piece of work [FS01, FLL+02], Flanagan et al, showed the feasibility of applying an extended static checker (named ESC/Java) to Java. Since then, several other similar systems have been further developed, including Spec#'s and its automatic

verifier Boogie [BLS04] that is applicable to the C# language. We adopt the same idea of allowing programmers to specify properties about each function (in the Haskell language) with contract annotations, but also allow contract annotations to be selectively omitted where desired. Furthermore, unlike previous approaches based on verification condition (VC) generation which rely solely on a theorem prover to verify, we use an approach based on symbolic evaluation that can better capture the intended semantics of a more advanced lazy functional language. With this, our reliance on the use of theorem provers is limited to smaller fragments that involve the arithmetical parts of expressions. Symbolic evaluation gives us much better control over the process of the verification where we have customised sound and effective simplification rules that are augmented with counter-example guided unrolling. More importantly, we are able to handle specifications involving recursive functions and/or higher-order functions which are not supported by any known automatic verification tools including ESC/Java and Spec#.

11.3 Dependent Type Approach

In the functional language community, type systems have played significant roles in guaranteeing better software safety. Advanced type systems, such as dependent types, have been advocated to capture stronger properties. While full dependent type system (such as Cayenne [Aug98]) is undecidable in general, Xi and Pfenning [XP99] have designed a smaller fragment based on indexed objects drawn from a constraint domain $\mathcal{C}$ whose decidability closely follows that of the constraint domain. Typical examples of objects in $\mathcal{C}$ include linear inequalities over integers, boolean constraints, or finite sets. In a more recent Omega project [She04], Sheard shows how extensible kinds can be built to provide a more expressive dependent-style system. In comparison, our approach is more programmer friendly as we allow arbitrary functions to be used in the contract annotations without the need to encode them as types. It is also easier for programmers to add properties incrementally. Moreover, our symbolic evaluation is formulated to adhere to lazy semantics and is guaranteed to terminate when code safety is detected or when a preset bound on the unrollings of each recursive function is reached. Compared with the dependent type approaches [XP99, CDX05, She04, WSW05] in general, we separate type and contract declarations so that type related work (e.g. type inference) and contract related techniques can be developed independently.

11.4 QuickCheck, Cover and Programatica Projects

Amongst the Haskell community, there have been several works that are aimed at providing high assurance software through validation (testing) [CH03], program verification [HJK+05] or a combination of the two [DQT03]. Our work is based on program verification. Compared to the Programatica project which attempts to define a P-Logic for verifying Haskell programs, we use Haskell itself as the specification language and rely on sound symbolic evaluation for its reasoning. Our approach eliminates the effort of inventing and learning a new logic together with its theorem prover. Furthermore, our verification approach does not conflict with the validation assisted approach used by

[CH03, DQT03] and can play complementary roles. We give a brief description of each project below.

QuickCheck [CH03] is an automatic random testing tool for Haskell programs. It is made up of a combinator library written in Haskell for an embedded specification language with test data generation and test execution. Program properties may be specified by either pre/post, algebraic style or model-based (functions or relations), before the properties are tested by random input generation. Given a function `f` which satisfies a property specification $\forall x \in A.P[x, \mathtt{f}(x)]$ with decidable property P, QuickCheck randomly generates values for x a preset number of times for its validation. It reports failure when a counter-example is found. QuickCheck could provide a profile of test generation to classify the proportion of trivial versus non-trivial tests. It allows user-specified random test data generators which is important for more complex data structures. A follow-up work for QuickCheck[CH03] has extended the random testing framework to cover monadic programs, where monadic properties are specified and then validated based on the notion of observational equivalence. As mentioned in Section 10.4, we can verify lemmas in QuickCheck's library in near future.

In the Cover project [DQT03], the idea of combining testing and proving is proposed for improving confidence in the correctness of Haskell programs. The aim of this combination is to harness the strengths of both testing and proving to provide a more effective way for ensuring the correctness of programs. Testing may be used for debugging programs and specification before a costly proof attempt. During a proof development, testing can quickly eliminate wrong conjectures. Proving helps to decompose a program into smaller components that could benefit from testing. This allows us a way to decompose the testing task. Current proof assistant technology requires great effort of a user, even for moderately complex program. Cover project intends to leverage on the lightweight nature of testing to handle bigger programs, and leaving only the more critical task to their Agda/Alfa proof assistant.

The Programatica project [HJK+05] aims to provide a sophisticated programming environment for the development of high assurance software system. Its components include a semantically rich, formal modelling language (Haskell), an expressive programming logic, called P-Logic, to capture critical program properties and a toolset for creating, maintaining and auditing the supporting evidences for different levels of software assurance. Program properties are expected for different stages of software development and can help validate code and also provide a good source of documentation. Both random testing using QuickCheck and proof assistant using Alfa (based on constructive type theory) are currently used for property validation.

11.5 Specification Inference

In [MR05, MR08], pattern matching failures can be inferred during compile time without specifications from programmers. Notably they use a language of regular expressions for contracts, which is incomparable with ours, and the technical details are very different. Moreover, high-order functions are inlined before analysis is applied in [MR05, MR08] while we analyse higher-order functions directly and do not lose modularity.

I myself have done some research in sized type (i.e. dependent type used in DML) inference for a first-order functional language [CKX01] and an imperative language [PXC08]. Recently, there is a piece of work on *liquid type* [RKJ08] which is on inferring dependent type in the presence of higher-order functions, by requiring only a set of logic qualifiers to be annotated. However, there are always some advanced properties that cannot be inferred. For example, the `noT1` property mentioned in Section 2.1.1 cannot be inferred by the liquid type framework because the `noT1` specifies a property that is true recursively for a data structure.

11.6 Logical Framework

In [HY04], a compositional assertion checking framework has been proposed with a set of logical rules for handling higher order functions. Given arguments satisfying their precondition, they check whether function definition satisfies its postcondition and the checking is currently a manual proof based on the logical rules. Apart from our focus on automatic verification, we can give precise blame when a contract violation is detected. The work in [HY04] has the strength in verification, but not in assigning blames.

11.7 Model for Dynamic Contract Checking

In [FF02], Findler and Felleisen introduce the notion of higher-order contracts. Later one, Blume and McAllester [BM06] introduce the concept of contract semantics and prove the soundness and completeness of the dynamic contract checking algorithm in [FF02] with respect to the contract semantics. We use a similar approach, but aimed at static checking. We use a lazy language, and support constructor contracts; on the other hand they deal with recursive contracts which we do not. Our grand theorem (Theorem 9) is close to their soundness and completeness theorems. However, the proof techniques are different. We use a crucial fact that the constructors $\triangleright$ and $\triangleleft$ form a projection pair while Blume and McAllester use a bisimulation strategy.

Also inspired by [FF02, BM06], Hinze et. cl. [HJL06] implement contracts as a library in Haskell and also provide contract constructors such as pairs, lists, etc. Compared with [FF02, BM06, HJL06], we apply a theory similar to those but to static contract checking so that we can detect bugs early. Another work on dynamic contract checking is the Camila project [VOS+05], which use monads to encapsulate the pre/post-conditions checking behaviour.

In [FB06], Findler and Blume discover that contracts are pairs of projections. Our projections use crashes-more-often as the partial ordering rather than the specificness of the contract. This difference partly due to the fact that we assume program itself may contain errors while they assume the only errors that can occur are due to contract violation. Moreover, we also discover the *projection pair* property which plays a crucial role in our proof.

11.8 Hybrid Type Checking and Hoare Type Theory

The static contract checking part of the hybrid contract checking framework [Fla06, KTG+06, KF07, GF07], in theory, can be as powerful as our system in terms of verifying program properties. But in practice, our symbolic execution strategy adopted from [Xu06] gives more flexibility to the verification as illustrated in Section 2.1.3 and also precise blames. In [WF07], Wadler and Findler show how contracts fit with hybrid types and gradual types by requiring casts in the source code. The casts are similar to our $\triangleright$ and $\triangleleft$. Compared with [Fla06, KTG+06, KF07, GF07, WF07], we deal with a lazy language and also handle constructor contracts.

In Hoare Type Theory (HTT) [NMB06, NAMB07], dependent types and a Hoare-style logic are combined for a language with higher-order functions and imperative commands. The approach can discover some bugs, but not all bugs while our approach can detect all bugs. We are also better at assigning blames to functions. But our contracts do not contain quantifiers while their Hoare types do.

11.9 Extended Static Checking for Haskell

Our first attempt to verify Haskell program is illustrated in [Xu06]. Later on, we realise that the checking code constructed is *strict* on contracts. For example, given:

```
f x y @ requires { x > 0 && y > 0}
f x y = case y <= 0 of
          True  -> error "f"
          False -> case x <= 0 of
                     True  -> error "f"
                     False -> 1
```

In [Xu06], the function `f#` corresponds to (but is not the same as) $f \triangleleft t_f$ and `f#` is:

$$
\begin{array}{l}
\lambda x.\lambda y.\ \texttt{case}\ x > 0\ \&\&\ y > 0\ \texttt{of} \\
\quad \texttt{False} \rightarrow \texttt{BAD} \\
\quad \texttt{True} \rightarrow\ \texttt{case}\ y \leq 0\ \texttt{of} \\
\qquad\qquad \texttt{True} \rightarrow \texttt{UNR} \\
\qquad\qquad \texttt{False} \rightarrow\ \texttt{case}\ x \leq 0\ \texttt{of} \\
\qquad\qquad\qquad\qquad \texttt{True} \rightarrow \texttt{UNR} \\
\qquad\qquad\qquad\qquad \texttt{False} \rightarrow 1
\end{array}
$$

We can see that the precondition is checked before the whole definition of the function. If x takes argument `bot` and y takes argument `-2`, `(f# bot -2)` diverges while the original code `(f bot -2)` crashes. This counter example shows that `f#` is not lazy enough.

On the other hand, in the contract approach, `f` has the following contract:

```
{-# CONTRACT f :: {x | x > 0} -> {y | y > 0} -> Ok
```

and its representative code $f \triangleleft t_f$ is:

$$
\begin{array}{l}
\texttt{f} \triangleleft \{x \mid x > 0\} \to \{y \mid y > 0\} \to \texttt{Ok} \\
= \left(\begin{array}{l} \lambda x.\lambda y.\texttt{case } y \leq 0 \texttt{ of} \\ \quad \texttt{True} \to \texttt{UNR} \\ \quad \texttt{False} \to \texttt{case } x \leq 0 \texttt{ of} \\ \qquad\qquad \texttt{True} \to \texttt{UNR} \\ \qquad\qquad \texttt{False} \to 1 \end{array} \right) \left(\begin{array}{l} \texttt{case } x > 0 \texttt{ of} \\ \texttt{True} \to x \\ \texttt{False} \to \texttt{BAD} \end{array} \right) \left(\begin{array}{l} \texttt{case } y > 0 \texttt{ of} \\ \texttt{True} \to y \\ \texttt{False} \to \texttt{BAD} \end{array} \right) \\
= \ldots \\
= \begin{array}{l} \lambda x.\lambda y.\texttt{case } y \leq 0 \texttt{ of} \\ \quad \texttt{True} \to \texttt{BAD} \\ \quad \texttt{False} \to \texttt{case } x \leq 0 \texttt{ of} \\ \qquad\qquad \texttt{True} \to \texttt{BAD} \\ \qquad\qquad \texttt{False} \to 1 \end{array}
\end{array}
$$

We can see that the contract of each parameter is checked just before it is used in the definition of `f`. That means we check the contract *lazily*.

11.10 Counter-example Guided Approach

Counter-example guided heuristics have been used in many projects (in which we can only cite a few) [BR02, HJM03] primarily for abstraction refinement. To the best of our knowledge, this is the first time it is used to guide unrolling which is different from abstraction refinement.

Chapter 12

Conclusion

We have presented a static contract checker for an advanced functional programming language, Haskell. With static contract checking, more bugs can be detected at compile-time and meaningful error messages can be reported early. We have demonstrated via examples the expressiveness of contracts and highlighted the effectiveness of our verification techniques. Apart from the fact that contract annotation helps in finding bugs, it is also a good form of documentation.

The main job of the static contract checker is to tell programmers where the bug is and why it is a bug. Unlike theorem provers, it is designed for ordinary programmers. But it has potential to be extended to prove more sophisticated properties. Contracts also have good potential for program optimisation to remove redundant runtime tests and unreachable dead code.

Our approach is sound as our symbolic evaluation follows closely the lazy semantics of Haskell. We have proved the soundness of each simplification rule and given a proof of the soundness of the static contract checking.

We extend our methodology to accommodate parametric polymorphism. Full Haskell (including type classes, IO Monad, etc) are transformed to the GHC Core Language [Tea98], which is in turn translated to the language $\mathcal{H}$. This translation makes the integration of the verification system into the Glasgow Haskell Compiler (GHC) easily. After we resolve all engineering problems and finish the enhancements in Chapter 10, GHC will be the first verifying compiler for an advanced functional language.

Bibliography

[Apt81] Krzysztof R. Apt. Ten years of Hoare's logic: A survey. *ACM Trans. Program. Lang. Syst.*, 3(4):431–483, 1981.

[Aug98] Lennart Augustsson. Cayenne - language with dependent types. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 239–250, New York, NY, USA, 1998. ACM Press.

[BCC+03] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Leino, and E. Poll. An overview of JML tools and applications, 2003.

[BLS04] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. *CASSIS*, LNCS 3362, 2004.

[BM06] Matthias Blume and David McAllester. Sound and complete models of contracts. *J. Funct. Program.*, 16(4-5):375–414, 2006.

[BR02] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3, New York, NY, USA, 2002. ACM Press.

[CDX05] Sa Cui, Kevin Donnelly, and Hongwei Xi. ATS: A language that combines programming with theorem proving. In Bernhard Gramlich, editor, *FroCos*, volume 3717 of *Lecture Notes in Computer Science*, pages 310–320. Springer, 2005.

[CH03] Koen Claessen and John Hughes. Specification-based testing with quickcheck. In *Fun of Programming*, Cornerstones of Computing, pages 17–40. Palgrave, March 2003.

[CH06] Olaf Chitil and Frank Huch. A pattern logic for prompt lazy assertions in Haskell. In Zoltán Horváth, Viktória Zsók, and Andrew Butterfield, editors, *IFL*, volume 4449 of *Lecture Notes in Computer Science*, pages 126–144. Springer, 2006.

[CH07] Olaf Chitil and Frank Huch. Monadic, prompt lazy assertions in Haskell. In Zhong Shao, editor, *APLAS*, volume 4807 of *Lecture Notes in Computer Science*, pages 38–53. Springer, 2007.

[CKX01] Wei-Ngan Chin, Siau-Cheng Khoo, and Dana N. Xu. Deriving pre-conditions for array bound check elimination. In Olivier Danvy and Andrzej Filinski, editors, *PADO*, volume 2053 of *Lecture Notes in Computer Science*, pages 2–24. Springer, 2001.

[CL07] Olaf Chitil and Yong Luo. Structure and properties of traces for functional programs. *Electr. Notes Theor. Comput. Sci.*, 176(1):39–63, 2007.

[CMR03] Olaf Chitil, Dan McNeill, and Colin Runciman. Lazy assertions. In Philip W. Trinder, Greg Michaelson, and Ricardo Pena, editors, *IFL*, volume 3145 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2003.

[Dav97] Rowan Davies. Refinement-type checker for Standard ML. In *AMAST '97: Proceedings of the 6th International Conference on Algebraic Methodology and Software Technology*, pages 565–566, London, UK, 1997. Springer-Verlag.

[DJ84] Werner Damm and Bernhard Josko. A sound and relatively* compete axiomatization of Clarke's Language L4. In *Proceedings of the Carnegie Mellon Workshop on Logic of Programs*, pages 161–175, London, UK, 1984. Springer-Verlag.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[DQT03] Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. Verifying Haskell programs by combining testing and proving. In *Proceedings of Third International Conference on Quality Software*, pages 272–279. IEEE Press, 2003.

[EMC77] Jr. Edmund Melson Clarke. Programming language constructs for which it is impossible to obtain good Hoare-like axiom systems. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 10–20, New York, NY, USA, 1977.

[FB06] Robert Bruce Findler and Matthias Blume. Contracts as pairs of projections. In *Functional and Logic Programming*, pages 226–241. Springer Berlin/Heidelberg, 2006.

[FBF06] R. B. Findler, M. Blume, and M. Felleisen. An investigation of contracts as projections. Technical report, University of Chicago Computer Science Department, 2006. Technical Report TR-2004-02.

[FF02] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 48–59, New York, NY, USA, 2002. ACM Press.

[Fla06] Cormac Flanagan. Hybrid type checking. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 245–256, New York, NY, USA, 2006. ACM Press.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM Press.

[FP91] Tim Freeman and Frank Pfenning. Refinement types for ML. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 268–277, New York, NY, USA, 1991. ACM Press.

[FPST07] Seth Fogarty, Emir Pasalic, Jeremy Siek, and Walid Taha. Concoqtion: indexed types now! In G. Ramalingam and Eelco Visser, editors, *PEPM*, pages 112–121. ACM, 2007.

[FS01] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 193–205, New York, NY, USA, 2001. ACM Press.

[FyGR07] Robert Bruce Findler, Shu yu Guo, and Anne Rogers. Lazy contract checking for immutable data structures. In Olaf Chitil, Zoltán Horváth, and Viktória Zsók, editors, *IFL*, volume 5083 of *Lecture Notes in Computer Science*, pages 111–128. Springer, 2007.

[GCH89] S. M. German, E. M. Clarke, and J. Y. Halpern. Reasoning about procedures as parameters in the language L4. *Inf. Comput.*, 83(3):265–358, 1989.

[GF07] Jessica Gronski and Cormac Flanagan. Unifying hybrid types and contracts. In *Eighth Symposium on Trends in Functional Programming*, April 2007.

[Goe85] Andreas Goerdt. A Hoare calculus for functions defined by recursion on higher types. In *Proceedings of the Conference on Logic of Programs*, pages 106–117, London, UK, 1985.

[GSSKT06] Jürgen Giesl, Stephan Swiderski, Peter Schneider-Kamp, and René Thiemann. Automated termination analysis for Haskell: From term rewriting to programming languages. In Frank Pfenning, editor, *RTA*, volume 4098 of *Lecture Notes in Computer Science*, pages 297–312. Springer, 2006.

[HJK+05] James Hook, Mark Jones, Richard Kieburtz, John Matthews, Peter White, Thomas Hallgren, and Iavor Diatchki. Programatica. *http://www.cse.ogi.edu/PacSoft/projects/programatica/bodynew.htm*, 2005.

[HJL06] Ralf Hinze, Johan Jeuring, and Andres Löh. Typed contracts for functional programming. In *FLOPS '06: Functional and Logic Programming: 8th International Symposium*, pages 208–225, 2006.

[HJM03] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Counterexample-guided control. *Automata, Languages and Programming: 30th International Colloquium, (ICALP03)*, 2719:886–902, 2003.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[HY04] Kohei Honda and Nobuko Yoshida. A compositional logic for polymorphic higher-order functions. In *PPDP '04: Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 191–202, New York, NY, USA, 2004. ACM Press.

[JVWW06] Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In Reppy and Lawall [RL06], pages 50–61.

[KF07] Kenneth Knowles and Cormac Flanagan. Type reconstruction for general refinement types. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007*. Springer-Verlag, April 2007.

[KTG+06] Kenneth Knowles, Aaron Tomb, Jessica Gronski, Stephen N. Freund, and Cormac Flanagan. SAGE: Unified hybrid checking for first-class types, general refinement types, and dynamics (extended report). *http://sage.soe.ucsc.edu/sage-tr.pdf*, 2006.

[LJBA01] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *POPL*, pages 81–92, 2001.

[LN98] K. Rustan M. Leino and Greg Nelson. An extended static checker for Modular-3. In *CC '98: Proceedings of the 7th International Conference on Compiler Construction*, pages 302–305, London, UK, 1998. Springer-Verlag.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall International, London, 1992.

[MFF06] Philippe Meunier, Robert Bruce Findler, and Matthias Felleisen. Modular set-based analysis from contracts. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL*, pages 218–231. ACM, 2006.

[MR05] Neil Mitchell and Colin Runciman. Unfailing Haskell: A static checker for pattern matching. In *TFP '05: The 6th Symposium on Trends in Functional Programming*, pages 313–328, 2005.

[MR08] Neil Mitchell and Colin Runciman. Not all patterns, but enough: an automatic verifier for partial but sufficient pattern matching. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 49–60, New York, NY, USA, 2008. ACM.

[MTH89] Robin Milner, Mads Tofte, and Robert Harper. The definition of Standard ML. 1989.

[NAMB07] Aleksandar Nanevski, Amal Ahmed, Greg Morrisett, and Lars Birkedal. Abstract predicates and mutable ADTs in Hoare Type Theory. In Rocco De Nicola, editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 189–204. Springer, 2007.

[NMB06] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in hoare type theory. In Reppy and Lawall [RL06], pages 62–73.

[ORY01] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL '01: Proceedings of the 15th International Workshop on Computer Science Logic*, pages 1–19, London, UK, 2001. Springer-Verlag.

[Par72] D. L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, May 1972.

[Pey96] Simon L. Peyton Jones. Compiling Haskell by program transformation: A report from the trenches. In *Proc European Symposium on Programming (ESOP)*, pages 18–44, 1996.

[PXC08] Corneliu Popeea, Dana N. Xu, and Wei-Ngan Chin. A practical and precise inference and specializer for array bound checks elimination. In Robert Glück and Oege de Moor, editors, *PEPM*, pages 177–187. ACM, 2008.

[Rey02] J. Reynolds. Separation logic: a logic for shared mutable data structures. *Invited Paper, LICS'02*, 2002.

[RKJ08] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 159–169, New York, NY, USA, 2008. ACM.

[RL06] John H. Reppy and Julia L. Lawall, editors. *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*. ACM, 2006.

[Sco76] D. S. Scott. Data types as lattices. *Society of Industrial and Applied Mathematics (SIAM) Journal of Computing*, 5(3):522–586, 1976.

[Ser07] Damien Sereni. Termination analysis and call graph construction for higher-order functional programs. In Ralf Hinze and Norman Ramsey, editors, *ICFP*, pages 71–84. ACM, 2007.

[She04] Tim Sheard. Languages of the future. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 116–119, New York, NY, USA, 2004. ACM Press.

[SJ05] Damien Sereni and Neil D. Jones. Termination analysis of higher-order functional programs. In Kwangkeun Yi, editor, *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 281–297. Springer, 2005.

[Tea98] The GHC Team. *The Glasgow Haskell Compiler User's Guide*. www.haskell.org/ghc/documentation.html, 1998.

[tea06a] The Coq team. The Coq proof assistant. *coq.inria.fr*, 2006.

[tea06b] The Isabelle/HOL team. Isabelle/HOL. 2006.

[VOS+05] J Visser, J. N. Oliveira, Barbosa L. S., J. F. Ferreira, and A. Mendes. CAMILA revival: VDM meets Haskell. In Nico Plat and Peter Gorm Larsen, editors, *Overture Workshop (co-located with FM'05)*, 2005.

[WF07] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *Workshop on Scheme and Functional Programming*, Sept 2007.

[WSW05] Edwin M. Westbrook, Aaron Stump, and Ian Wehrman. A language-based approach to functionally correct imperative programming. In Olivier Danvy and Benjamin C. Pierce, editors, *ICFP*, pages 268–279. ACM, 2005.

[XCC03] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *POPL*, pages 224–235, 2003.

[Xi99] Hongwei Xi. Dead code elimination through dependent types. In Gopal Gupta, editor, *PADL*, volume 1551 of *Lecture Notes in Computer Science*, pages 228–242. Springer, 1999.

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 214–227, New York, NY, USA, 1999. ACM Press.

[Xu06] Dana N. Xu. Extended static checking for Haskell. In *Haskell '06: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 48–59, New York, NY, USA, 2006. ACM Press.

Appendix A

Deletion for AVL Tree

In this section, we give an incorrect version of deletion. We claim that we can verify those correctly defined branches and detect the incorrectly defined branches marked by (Wrong).

```
-- balanced deletion
{-# CONTRACT delete :: {x | balanced x} -> Ok ->
                       {r | balanced r && 0 <= depth x - depth r &&
                            depth x - depth r <= 1} #-}
delete :: AVL -> Int -> AVL
delete v1 v2
= case v1 of
   L -> L
   N v L L -> case v == v2 of
               True  -> L
                 False -> N v L L
   N v t L -> case v == v2 of
               True  -> t
               False -> let t1 = delete t v2
                        in N v t1 L
   N v L u -> case v == v2 of
               True -> u
               False -> N v L (delete u v2)
   N v t u ->
    case v == v2 of
     True -> case depth t > depth u of
              True  -> let t1 = delete t (btmax t)
                       in N (btmax t) t1 u   -- B2
              False -> let u1 = delete u (btmin u)
                      in  N (btmin u) t u1
     False -> case v2 < v of
               True -> let t1 = delete t v2
```

```
                                t2 = insert t1 v
                                u1 = delete u (btmin u)
                            in case depth u - depth t1 > 1 of
                                True -> case depth u1 - depth t2 <= 1 of
                                            True -> N (btmin u) t2 u1
                                          False -> case u1 of
                                                     N x l r ->
                                                      case depth r > depth l of
                                                        True  -> -- (Wrong)
                                                                 lrotate (N v t2 u1)
                                                         False -> N (btmin u) t2 u1)
                                   False -> N v t1 u
                         False -> let u1 = delete u v2
                                      u2 = insert u1 v
                                      t1 = delete t (btmax t)
                                  in case depth t - depth u1 > 1 of
                                      True -> case depth t1 - depth u2 <= 1 of
                                                  True -> N (btmin t) u2 t1
                                             False -> case t1 of
                                                        N x l r ->
                                                         case depth r > depth l of
                                                           True  -> -- (Wrong)
                                                                    lrotate (N v u2 t1)
                                                            False -> N (btmax t) t1 u2)
                                   False -> N v t u1

max :: Int -> Int -> Int
max x y = case x >= y of
           True -> x
           False -> y

btmax :: AVL -> Int
btmax (N x _ L) = x
btmax (N x t u) = btmax u

btmin :: AVL -> Int
btmin (N x L _) = x
btmin (N x t u) = btmin t
```

Appendix B

Insertion for Red-Black Tree

A red-black tree is a type of self-balancing binary search tree, which was invented by Rudolf Bayer in 1972. Like AVL tree, it can also search, insert and delete in $O(log\ n)$ where n is the number of the node in the tree prior to the operation. In a red-black tree, the leaf nodes are not relevant and do not contain data. A red-black tree is a binary search tree where each node has a colour attribute, the value of which is either *red* or *black*. In addition to the ordinary requirements imposed on binary search trees, the following additional requirements of any valid red-black tree apply:

(1) A node is either red or black.

(2) The root is black.

(3) All leaves are black, even when the parent is black.

(4) Every red node has two black children, with `E` being regarded black as well.

(5) Every simple path from a node to a descendant leaf contains the same number of black nodes, either counting or not counting the null black nodes.

In this section, we claim that our system can verify Okasaki's insertion algorithm, maintains the invariant (5) slavishly. However, the invariant (4) is slightly weakened: read node at the root of a tree may have red children. We call such red-black trees "infrared", the other red-black trees "proper".

```
module RedBlack where

data Colour = R | B deriving Show
data RB = E | T Colour RB Int RB deriving Show

{- Insertion and membership test as by Okasaki -}
rbinsert ::  RB -> Int -> RB
{-# CONTRACT rbdelete :: {s | rbinvariant t} -> Ok -> {r | rbinvariant r} #-}
```

```
rbinsert s x = makeBlack (ins s x)

{-# CONTRACT ins :: {s | rbinvariant t} -> Ok -> {r | rbinvariant r} #-}
ins t x
 = case t of
    E -> T R E x E
    T color a y b -> case x < y of
                     True -> balance color (ins a x) y b
                     False -> case x == y of
                              True -> t
                             False -> balance color a y (ins b x)

makeBlack (T _ a y b) = T B a y b

{- balance: first equation is new,
   to make it work with a weaker invariant -}
balance :: Color -> RB -> Int -> RB -> RB
-- balance B (T R a x b) y (T R c z d) = T R (T B a x b) y (T B c z d)
balance B (T R (T R a x b) y c) z d = T R (T B a x b) y (T B c z d)
balance B (T R a x (T R b y c)) z d = T R (T B a x b) y (T B c z d)
balance B a x (T R (T R b y c) z d) = T R (T B a x b) y (T B c z d)
balance B a x (T R b y (T R c z d)) = T R (T B a x b) y (T B c z d)
balance a x b = T c a x b

-- invariant: (4)
redinvariant E = True
redinvariant (T R (T R _ _ _) _ _) = False
redinvariant (T R _ _ (T R _ _ _)) = False
redinvariant (T _ l _ r) = redinvariant l && redinvariant r

blackdepth E = 0 -- we don't count Es as real nodes
                 -- if they were, they would be black
blackdepth (T B a _ _) = 1 + blackdepth a
blackdepth (T R a _ _) = blackdepth a

-- invariant: (5)
blackinvariant E = True
blackinvariant (T _ a _ b) = blackdepth a == blackdepth b &&
                             blackinvariant a && blackinvariant b

rbinvariant t = redinvariant t && blackinvariant t
```

Appendix C

Programs in Experiments

Total functions from Haskell library can be used (e.g. not, null, etc) as the library is compiled with -O option, which means the library functions can be inlined. Here, we have rewritten some library functions (e.g. and2) for some basic testings. In our current implementation, we use the notation `_` for contract `Any`.

C.1 List

```
and2 True x = x
and2 False x = False

{-# CONTRACT head1 :: {xs | not (null xs)} -> {z | True} #-}
head1 :: [Int] -> Int
head1 (x:xs) = x

{-# CONTRACT tail1 :: {xs | not (null xs)} ->  {z | True} #-}
tail1 :: [Int] -> [Int]
tail1 (x:xs) = xs

{-# CONTRACT head2 :: {xs | not (null xs)} -> {z | True} #-}
head2 :: [[Int]] -> [Int]
head2 (x:xs) = x

-- {-# CONTRACT headtail :: {x | not (null (head2 x))} ->  {z | True} #-}
-- to show the importance of giving precondition for head2
{-# CONTRACT headtail :: {x | and2 (not (null x)) (not (null (head2 x)))}
                      ->  {z | True} #-}
headtail :: [[Int]] -> [Int]
headtail (y:ys) = tail1 y

-- with this contract, the caller of res2
```

```
-- may be considered as a non-safe function
{-# CONTRACT res2 :: {x | True} ->  _ #-}
res2 x = head1 [1]

res2_crash = res2   -- unsafe because res2's postcondition is Any

{-# CONTRACT res3 :: {x | True} #-}
res3 = head1 [1]

res4 = (head1 [6], head1 [])   -- unsafe

res5 = head1 [1]

res6 x = case x of
         True ->  res4   -- unsafe
         False -> (res5, 1)

{-# CONTRACT last1 :: {ys | not (null ys)} -> {z | True} #-}
last1 :: [Int] -> Int
last1 [x]    = x
last1 (x:xs) = last1 xs

{-# CONTRACT negList :: {ys | not (null ys)} -> {z | True} #-}
negList [x]    = [not x]
negList (x:xs) = (not x) : negList xs

{-# CONTRACT minimum1 :: {xs | not (null xs)} -> {z | True} #-}
minimum1 :: [Int] -> Int
minimum1 [x]        = x
minimum1 (x1:x2:xs) = let m = case (x1 < x2) of
                              True -> x1
                              False -> x2
                      in minimum1 (m : xs)
```

C.2 Tuple

```
{-# CONTRACT f0 :: {a | not (null a)} -> {z | True} #-}
f0 :: [Bool] -> Bool
f0 (x:xs) = x

-- This is a stronger contract which requires the argument to be nonNull.
{-# CONTRACT f :: {a | not (null a)} -> ({y | True}, {z | True}) #-}
f :: [Bool] -> (Bool, Bool)
```

```
f x = (True, f0 x)


{-# CONTRACT fst2 :: ({x | True}, _) -> {z | True} #-}
fst2 :: (Bool, Bool) -> Bool
fst2 (a,b) = a

{-# CONTRACT snd2 :: (_, {x | True}) -> {z | True} #-}
snd2 :: (Bool, Bool) -> Bool
snd2 (a,b) = b

g1 x = fst2 (f x)

g2 x = snd2 (f x)

g3 x = f0 x

h [] = True
h (x:xs) = g2 xs
```

C.3 Higher-Order Functions

```
{-# CONTRACT f1 :: ({x | True} -> {y | >= 0}) -> {z | z >= 0} #-}
f1 :: (Int -> Int) -> Int
f1 g = g 1 - 1

f2 = f1 (\x -> x -1)
```

C.4 Data Type

```
-- non-recursive data type
data A = A1 | A2
data B = B1 | B2

noA2 A1 = True
noA2 A2 = False

yesA2 A1 = False
yesA2 A2 = True

{-# CONTRACT h1 :: {x | noA2 x} -> {z | yesA2 z} #-}
h1 :: A -> A
```

```
h1 A1 = A2


g1 :: A -> A
g1 A1 = A1
g1 A2 = A1

{-# CONTRACT h2 :: {x | not (noA2 x)} -> {z | not (yesA2 z)} #-}
h2 :: A -> A
h2 A2 = A1

test = h1 (g1 A2)  -- this is safe


-- f1, f2, f3 test error msg generation

{-# CONTRACT f1 :: {x | noA2 x} -> _ #-}
f1 :: A -> A
f1 x = h1 A2

f3 x y =  case y of
            B1 -> f2 x y

f2 x y = case y of
            B1 -> f1 x
```

C.5 Data Constructor Contract

```
{-# CONTRACT g1 :: ({x | x > 0}, _ ) -> {z | z > 0} #-}
g1 :: (Int, Int) -> Int
g1 (x,y) = x

{-# CONTRACT g2 :: ( _ , {y | y > 0}) -> {z | z > 0} #-}
g2 :: (Int, Int) -> Int
g2 (x,y) = y

bad = error "bad!"

t1 = g1 (5, bad)
t2 = g2 (bad, 6)
t3 = g1 (bad, 7)
t4 = g1 (-1, 6) -- seems inlining is done.

data A a = B a | C a Int
```

```
{-# CONTRACT g :: B {x | x > 0} -> _ #-}
g :: A Int -> Int
g (B x) = x
g (C x y) = y
```

C.6 Recursive Functions in Contracts

```
len []     = 0
len (x:xs) = 1 + len xs

-- contract with lenLeq1 works more efficiently than
-- contract with len function i.e. a recursive function

lenLeq1 []  = True
lenLeq1 [a] = True
lenLeq1 xs  = False

-- {-# CONTRACT f :: {xs | (len xs) <= 1} -> {r | True} #-}
{-# CONTRACT f :: {xs | lenLeq1 xs} -> {r | True} #-}
f :: [Int] -> Int
f [] = 0
f [x] = 1
```

C.7 Arithmetic

```
-- {-# CONTRACT f :: {x | x >= 0} -> {r | r < 1} #-}
{-# CONTRACT f1 :: {x | x >= 0} -> {r | True} #-}
f1 :: Int -> Bool
f1 x = x > 1


{-# CONTRACT f2 :: {x | True} -> {r | r == x + 1 } #-}
f2 :: Int -> Int
f2 x = x + 1


{-# CONTRACT g :: {x | True} #-}
g = f1 (-1)
```

```
-- {-# SPECIALISE f :: Int -> Int #-}
-- {-# CONTRACT f :: x:{y | y > 0} -> {r | r == x + 1} #-}
-- {-# CONTRACT f :: x:{y | y > 0} -> {y | y > 0} -> {r | r == x + 1} #-}
-- {-# CONTRACT f :: any -> {y | y > 0}  #-}
-- {-# CONTRACT f :: {y | y > 0} -> _ #-}
{-# CONTRACT inc :: {y | y > 0} -> {r | r > 1} #-}
inc :: Int -> Int
inc x = x + 1


-- {-# CONTRACT sum2 :: x:{y | y > 0} -> {y | y > x} -> {r | r > 0} #-}
sum2 :: Int -> Int -> Int
sum2 x y = x + y


{-# CONTRACT mul :: {x | x >= 0} -> {y | y >= 0} -> {r | r >= 0} #-}
mul :: Int -> Int -> Int
mul x y = x * y

t1  = inc 5
t2a = sum2 (inc 5) 2
t2b = sum2 (inc 5) 6
```

Glossary

$\rightarrow_M$	expression reduction with fuel M	42
$e_1 \equiv_s e_2$	expression e_1 is semantically equivalent to expression e_2	42
$t_1 \equiv_t t_2$	contract t_1 is equivalent to contract t_2	60
$\vdash \mathcal{C}[\![e]\!] :: ()$	expression $\mathcal{C}[\![e]\!]$ is closed and well-typed	45
$e \in t$	expression e satisfies contract t	53
$e \triangleright t$	e ensures t	63
$e \triangleleft t$	e requires t	63
$e_1 \preceq e_2$	e_1 crashes more often than e_2	48
$e_1 \ll_R e_2$	e_1 behaves the same as e_2 or throw exceptions in the set R	47
$\bowtie$	contract projection	64