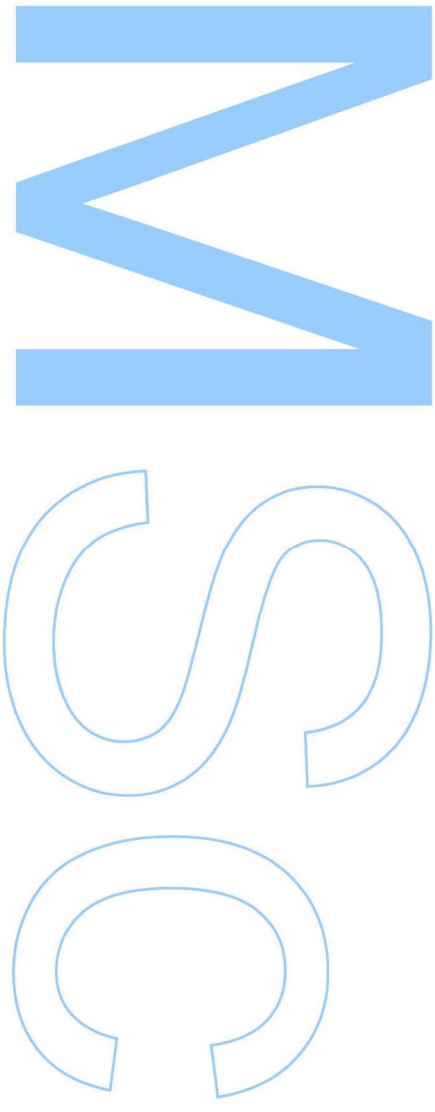
Contract Checking For Lazy Functional Languages

Rui Jorge Santos Andrade
Mestrado em Ciência de Computadores
Departamento de Ciência de Computadores
2018

Orientador
António Mário da Silva Marcos Florido, Professor Associado,
Faculdade de Ciências da Universidade do Porto



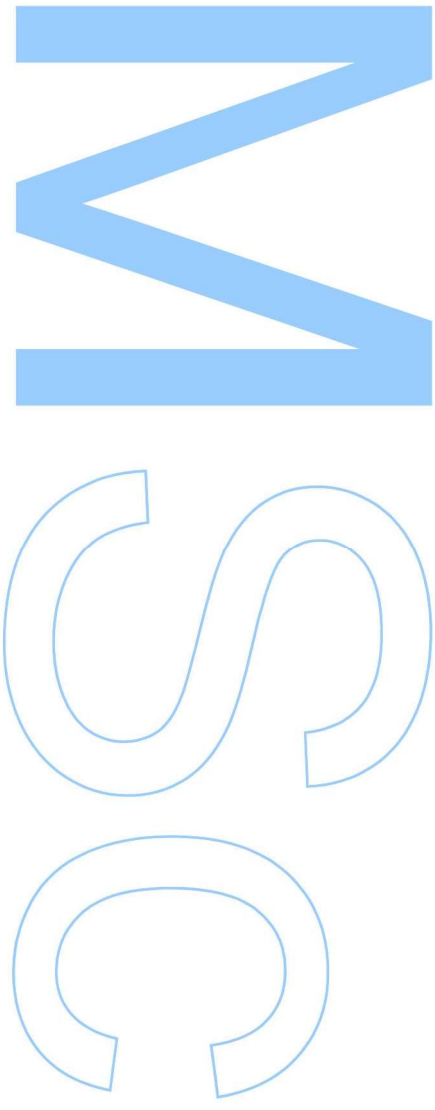
U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____



Abstract

Program errors are sometimes hard to find and program debugging is now a significant part of the program development cycle, leading to an increase of time and cost of program development. Several verification techniques were defined in the past, mostly for imperative and object-oriented programming languages. For pure functional programming languages, verification is apparently not so crucial, because programs tend to be more correct by design and by using the highly expressive type systems of these languages. However, exactly these declarative features of pure functional programming open the way to the verification of more sophisticated properties. In this thesis, we first give a short overview of two main verification frameworks for the Haskell programming language: Liquid Types and Contract Checking. We then describe one implementation of a static property checker based on program transformation. Thus, we have implemented in Haskell:

- A program transformation algorithm which enables to check if a program respects a given specification
- A parser for a contract language, using monadic parsing
- An inlining and simplification algorithm, which is based on beta-reduction
- A module which enables our system to interface with external theorem provers

In this framework, to tackle higher-order program verification, we use program inlining until properties are easily checked by several automatic theorem provers.

Resumo

Por vezes, erros em programas são difíceis de encontrar, e *debugging* de programas é uma parte fulcral do desenvolvimento de software, levando a um aumento do tempo e custo do desenvolvimento em si. No passado, estruturaram-se várias técnicas de verificação, quase todas para linguagens imperativas ou orientadas a objetos. Para programação funcional pura, não há uma preocupação tão grande, uma vez que normalmente os programas são correctamente escritos, auxiliados por sistemas de tipos altamente expressivos. Contudo, é a declaratividade da programação funcional pura que dá azo à possibilidade de verificação de propriedades mais sofisticadas sobre os programas. Nesta tese, efetuamos uma revisão e estudo de duas abordagens principais: Liquid Types e Verificação de Contratos. De seguida, descrevemos uma implementação de um sistema de verificação estática de propriedades baseado em transformação de programas. Em resultado do estudo, implementámos em Haskell:

- Um algoritmo de transformação de programas que permite verificar se um programa respeita a sua especificação
- Um *parser* para uma linguagem de contratos, baseado em *parsing* monádico
- Um algoritmo de *inlining* e simplificação, baseado em beta-redução
- Um módulo que permite ao nosso sistema comunicar com *theorem provers* externos

Este sistema faz uso de *inlining* de funções para que as propriedades sejam verificadas com ajuda de *theorem provers* externos, com vista a abordar o problema da verificação de programas de ordem superior.

Agradecimentos

A elaboração de uma dissertação é uma experiência paradoxal em várias áreas, mas quero destacar duas fundamentais. Olhando para a área académica, é a conjunção de uma boa parte do material que se aprende ao longo dos anos, mas é, também, um espaço de extravasação intelectual, na medida em que somos incentivados a investigar, a descobrir, a inventar, a tentar e falhar, a chegar a algo de novo, fugindo do que já é consolidado – mas sempre sem o descurar. Reflectindo sobre a área humana, é uma constante luta entre o "eu percebo algo disto, eu sei como posso juntar isto e aquilo para chegar ao meu objectivo" e "isto não está a dar, isto não vai funcionar, não me entendo com esta abordagem, isto é muito diferente".

Se na área académica o paradoxo faz mais sentido, uma vez que a tentativa e erro e o estudo são fulcrais para o desenvolvimento da mesma, na área mental é mais difícil racionalizar o pensamento que lhe corresponde, tantas vezes numa dissertação, como noutras situações pelas quais teremos, inevitavelmente, de passar. E por isso é que, à imagem do curso, uma dissertação não se faz sozinha – tanto no sentido de que requer um esforço consistente e constante, como no sentido de que convém sempre ter algum tipo de apoio ou acompanhamento.

Em suma...

Ao Prof. Mário Florido, que teve toda a disponibilidade que se podia pedir e mais alguma, e que acreditou em mim para levar avante este trabalho,

À minha família, que me atura todos os dias sem excepção, facilmente quando o trabalho vai bem e menos facilmente quando o trabalho não vai bem,

À Brenda, que me dá o prazer de a aturar e a quem eu dou um deleite semelhante, e que me viu a dar o último "passo" para conseguir finalizar este trabalho,

Aos meus amigos, sem qualquer excepção nem enumeração – até pelo risco de, inadvertidamente e acidentalmente, desconsiderar ou me esquecer de alguém –, pelas incontáveis horas de faculdade, de estudo, de discussão, de diversão, de convívio e, obviamente, de muito, muito, muito trabalho,

MUITO OBRIGADO.

Ao meu pai

Contents

List of Figures

Chapter 1

Introduction

Program verification is an important area in programming, with applications to simpler and more accurate debugging techniques, and easier bug fixing.

It is somewhat common to see developers implement assertions [1] as a method for checking whether a computing state is valid, however these do not suffice for certain cases.

It is possible to go a step further in program safety testing, with the help of contracts. These are based on boolean assertions and allow for defining more complex tests over the program state and for detailed blame assignments [2].

There are many areas where it is critical that code does not unexpectedly fail, including medicine or some real-time systems. Moreover, being able to understand where the fault is actually occurring and what part of the code is responsible for it, will surely make it simpler to correct problems and to write cleaner, more sound code as a result.

This thesis focuses on contract checking for functional programming, more specifically Haskell.

Certain aspects of functional programming languages make it harder to implement the most obvious verification techniques. For example, one could be tempted to merely test every argument passed to a function, but this could be a troublesome strategy when we have higher-order partially applied functions. Lazy evaluation can also be difficult to handle, in particular when we have recursive function calls or when the result is potentially diverging (e.g. an infinite list).

In this work, we implement a system for contract verification in Haskell and apply it to some program examples. We show how our approach allows for efficient and conclusive detection of potential faults in Haskell code.

Initially, we will survey existing solutions for program verification in functional languages, including static and dynamic contract verification and type-based verification.

After being aware of current developments in this area, we will select a subset of Haskell and extend it with contracts, through implementation of a compiler that transforms and simplifies programs, focusing on operational semantics derivations that would make them violate their

contracts. We have properties expressed in Haskell, so as to not force a developer to learn and use a different language, and to verify them with the help of external theorem provers.

Finally, we will test our proposed system with real Haskell programs and modules, drawing conclusions from the results we obtain.

Our contribution is a practical implementation of a verification system for Haskell based on previous work described in [3][4], ready for usage with examples that clearly illustrate its usefulness. The system has the following original contributions:

- Our transformation technique stops earlier than the system described in [3], producing more expressive verification conditions which are then sent to a theorem prover. In [3], properties are sent during the simplification process. Our approach is, therefore, more compositional, having the proving and simplification modules separate.
- We use more than one theorem prover, which can be useful for instances where satisfiability is not provable with certain approaches. In [3], Simplify [5] is used, but we also use Z3 [6].
- Our system is able to handle recursive functions in contracts, analyzing function calls to check if their arguments are constructors, and unrolling them in separate in that case. In [3], these programs are tested by hand, but our system can deal with them.

This thesis reports an implementation work. Formal correctness of the algorithms implemented here was presented in [3]. Our original extensions to that work are easily shown to be sound with respect to the original work, because we always use semantics-preservation program transformation techniques, such as inlining.

We assume that the reader is familiar with the Haskell programming language. Good references for Haskell include the textbook *Haskell: The Craft Of Functional Programming* by Simon Thompson [7] and the website `http://www.haskell.org`.

Our system allows a developer to test their code with minimal changes to it, mostly related to having to express which properties should hold at every given stage of a program. The implementation is available in `https://github.com/randrade23/sccHaskell`. The main results of this thesis were reported in [8].

In the next chapter, we describe the current state of the art in this area, detailing existing approaches to this problem. In chapter 3, we explain how we implement a verification system, including what is supported and how the system does its work. In chapter 3.2, we detail how program transformation is essential to performing verification. In chapter 4, we show how external theorem provers are used to aid in proving the safety of a program. In chapter 5 we present examples of verification done with the implemented system. Finally, we conclude and point some future lines of research.

Chapter 2

State of the art

Some previous approaches have been taken to program verification for functional languages, with certain ones more complete than others.

A first example of a static strategy to ensure that written code is adequate is the language's type system. ML and Haskell use Damas-Milner type inference [9], which refers only to the types of function arguments and not to the functions themselves. This prevents errors such as *1 + True*, but will not detect unsafe calculations such as division by zero, since it is only concerned with the types of values themselves. The type inference algorithm can be extended easily to test values with some arithmetic, as the work described in section 2.1 does.

Another approach which extends the previous one is *dependent types* [10]. Dependent types allow to parameterize a type according to a value – for example, the length of a list could be specified directly in the type, allowing for more restricted and effective type checking. Some functional programming languages implement dependent types as their type system, including IDRIS [11] and Agda [12] (see figure 2.1). Interactive theorem provers Coq [13] and Isabelle [14] also implement dependent type theory, allowing for verification based on such specifications. In this thesis, we opted to describe in more detail two approaches to functional program verification: liquid types and systems that are based on program transformations.

```
data _X_ (A B : Set) : Set where
    <_,_> : A -> B -> A X B

zip : {A B : Set} -> List A -> List B -> List (A X B)
zip [] [] = []
zip (x :: xs) (y :: ys) = < x , y > :: zip xs ys
zip _ _ = []
```

Figure 2.1: Expressing the Cartesian product with dependent types in Agda, and typing the *zip* function with it

Adrion *et al.* [15] divide program testing techniques into two categories: static and dynamic – verifying at compile-time or at run-time. Static verification can capture a great amount of potential errors, however, as we will see, it is not a complete solution. Dynamic verification has the advantage of being able to test with real program executions, but it makes programs take longer to execute, by actively testing them during their execution and it restricts itself to certain data flows only, which means it is also not a general purpose approach.

2.1 Verification using refinement types

Refinement types [16] allow us to define restrictions over a type using logical predicates. Testing whether or not a value is admissible in a refinement type is usually done by converting the restrictions to *verification conditions*, which are then passed on to a Satisfiability Modulo Theory (SMT) solver [17] that determines whether they are compatible with each other. However, this verification strategy is not sufficient on its own for what we intend to verify – as we will show, we also need to focus on the definition of the programs themselves.

Vazou *et al.* defined a type system – liquid types – which enables refinement types with automatic type checking and partial type inference from a set of initial refinement annotations. Liquid types were implemented as *LiquidHaskell* [18], which verifies the safety of Haskell code.

Example 2.1. A type for positive integers may be defined as such:

```
type Pos = {v:Int | v > 0}
```

Example 2.2. Defining a data structure for a CSV file, restricting the columns on each row to be the same as the number of columns defined in the header:

```
data CSV a = CSV { cols :: [String]
                 , rows :: [ListL a cols] }
type ListL a X = {v:[a]| len v = len X}
```

Predicates can be defined with constants, variables, expressions, comparisons and *measures*. Measures are Haskell functions that represent inductive properties and they are defined from a limited subset of the language, since we are concerned with termination and restricted to the expressivity of the SMT solver. It is clear that it is not possible to decide whether or not a program terminates, so care is taken in how Haskell code can be used to define properties. However, divergence can still occur in the code we are testing for safety, in particular when we have recursive functions. It is expected that recursive or looping functions have some termination metric associated to them, which allows LiquidHaskell to assume whether or not a function is approaching a final state. The usage of termination metrics is important, since it makes LiquidHaskell into a verifier of total correction. It becomes clear that this approach shares some similarities to a generic static verification approach, with some extensions, namely the concern

for termination. While termination checking is a big advantage of this approach, it is also less expressive in the definition of properties, given how the definition of measures is more restricted than, for example, the approach described in section 2.3.

2.1.1 Termination metrics

Termination metrics help a verifier deduce whether a function terminates or not. In our case, they are defined with measures, which represent natural numbers. The measure of arguments passed to a function must be smaller in every subsequent call to that function. Since measures are decreasing consistently, they are a particular case of the well-defined relation $(\mathbb{N}, <)$ meaning that they will decrease towards 0 – once this happens, the verifier knows that recursion has ended.

Example 2.3. Defining a measure for the length of a list:

```
len [] = 0
len (x:xs) = 1 + len xs
```

Example 2.4. Using the list length measure to type the *map* function:

```
map :: (a -> b) -> xs:[a] -> [b] / [len xs]
map f [] = []
map f (x:xs) = f x : map f xs
```

In the case of the *map* function, we use the length of the source list as the termination metric. The function will be called with lists of decreasing size, eventually reaching an empty list. In addition to the metric being well-defined, the base case does not make another recursive call, so it becomes clear that the function does terminate.

2.2 Dynamic verification

As discussed earlier, we can take an alternative approach to program verification by using a dynamic strategy, running during the execution of the program itself. Findler & Felleisen [19] defined an assertion-based contract verification system which supports higher-order functions and values, holding on to them until first-order values are produced and then testing them accordingly. The system was devised in the form of a typed lambda calculus, which wraps code in a recursive testing function that also assigns blame to the function caller or to the callee, depending on who produced the erroneous value.

An implementation of this system has been produced in Scheme, a strict functional programming language, with support for writing contracts in Scheme itself. During program

```
wrap :: Contract -> a -> Function -> Function -> a
wrap p e f f' = case flatContract p of
  True -> if p e then e else error f
  False ->
    let
      d = dom p
      r = rng p
    in
      \y -> wrap r (e (wrap d y f' f)) f f'
```

Figure 2.2: Pseudocode for the wrapping algorithm, as described by Findler & Felleisen

transformation, contracts are classified into two categories – flat contracts and function contracts. Flat contracts correspond to conditions over a first-order value and are tested by simply running the condition, eventually blaming who provided it – for example, if *main* called a function with an inadequate argument, the blame would lie with *main*. Function contracts result in another wrapping of the function, splitting the contract in two: one contract for the argument (the domain of the contract), which is verified as a flat contract, and another one for the result (the range of the contract), but also alternating the blame adequately – if testing the result fails, the blame lies with the function itself, instead of its caller. The following program gets the head of a list and prints it, with a contract for the *head* function to check whether the list is empty or not. The original program and the result of the transformation are presented below.

Example 2.5. Defining *head* with a contract for safety:

```
main = putStrLn $ show $ head [1,2,3]

{-@ head :: (not . null) -> ok @-}
head :: [a] -> a
head (x:xs) = x
```

Adding dynamic checks for whether *head*'s contract is violated or not:

```
main = putStrLn $ show $ head' [1,2,3]

head' :: [a] -> a
head'
  = \ y0
      -> (head (if (not . null) y0 then y0 else error "caller"))

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

The result of this transformation produces code which preserves type signatures for the original functions and that are executable without any need for the developer to change function calls or anything related to contracted functions. This sort of approach could be implemented in Haskell with the help of Template Haskell [20], a metaprogramming extension which allows for the generation of type-safe code on-the-fly.

Example 2.6. As another example, we present a definition of the Quicksort algorithm and its contract, as well as a possible redefinition with checking for contract violations and blame assignments.

```
{-@ quicksort :: ok -> sorted @-}
quicksort :: [Int] -> [Int]
quicksort [] = []
quicksort (x:xs) = quicksort [y | y <- xs, y <= x] ++ [x] ++
   quicksort [y | y <- xs, y > x]

quicksort'
      = \ y0
          -> if sorted
                  (quicksort (if ok y0 then y0 else error
                     "caller")) then
                quicksort (if ok y0 then y0 else error "caller")
             else
                error "callee"
      where
          quicksort [] = []
          quicksort (x : xs)
            = (quicksort [y | y <- xs, y <= x]) ++ [x]
              ++ quicksort [y | y <- xs, y > x]
```

The *quicksort* function is wrapped inside another function, which checks for condition violations. The *ok* function is a function that accepts any argument, thus representing that we do not have a condition to enforce. As such, the redefined function will not run any checks on the argument, but it will check if applying *quicksort* to a list does, indeed, produce a sorted list, throwing an error if not. This redefinition will blame itself if a postcondition violation is detected, meaning that it would be incorrectly defined. The original definition is preserved and used internally in this new function.

2.3 Static verification using program transformations

Inspired by the lack of verification tools for lazy functional languages, Xu *et al.* [4] developed a static verification system for Haskell which bases itself on the contracts provided to each function

and that uses Haskell to write those contracts. This work focused on as much of the language as possible, including higher-order functions or custom data structures, and integrated directly with GHC, testing intermediate code and stopping at that stage if any potential unsafety is detected. This strategy serves as a foundation for the system that we implement in this dissertation and has been proven to be sound [3].

For this approach, programs are transformed to include code that aids us in testing whether or not code should be considered safe. These transformations are then simplified and analyzed, to check whether the programs conform with their specifications. This means that the result of testing relies mostly with Haskell's semantics, with the exception being when we have arithmetic conditions – these are passed on to an SMT solver, similarly to the strategy described in section 2.1. In contrast to LiquidHaskell, however, conditions can be expressed with nearly as much expressivity as standard Haskell code, not being restricted in number of arguments, equations or even recursion.

Example 2.7. Given the definitions of $noA2$, $yesA2$, $h1$ and $g1$, and as a result of inlining and simplification, $h1$ and $test$ are proven safe, since they conform to their contracts or make safe function calls.

```
data A = A1 | A2

noA2 A1 = True
noA2 A2 = False

yesA2 A1 = False
yesA2 A2 = True

g1 :: A -> A
g1 A1 = A1
g1 A2 = A1

{- CONTRACT h1 ::  x | noA2 x -> z | yesA2 z -}
h1 :: A -> A
h1 A1 = A2

test = h1 (g1 A2)
```

As for termination, functions are called with a limited "fuel", which represents how many times we inline the definition of a function until we take it as final. This also means that diverging functions could, ultimately, be considered safe, which is arguable, but also a reasonable enough solution for a problem that is undecidable. While this serves as a reminder that contract verification is, in general, an undecidable problem, we can use certain heuristics to estimate, to some degree of certainty, whether or not a program should be considered safe. If we take, as an

example, the function *length*, it would be inlined infinitely if there were no such limit, since it is recursive. With this approach, it would branch out into two possibilities: 0 or $1 + lengthl$, which can be inlined as many times as it is necessary to get enough detail about the code we are analyzing. This strategy is explained and used in detail in some examples of chapter 5.

This is, indeed, sufficient for testing programs for partial correctness, which is the goal of this approach. For deciding if programs are safe, one technique used by Xu is checking, after simplification, whether or not a program is *syntatically safe*, meaning that it does not call directly a term that we know will fail.

One important contribution that we will make in this thesis, described in detail in the next chapters, is the possibility of inlining and taking advantage of the definition of recursive contracts. In the implementation detailed in [3], certain properties over functions that work with lists, such as *append*, are not proven safe or unsafe, due to the recursion in its specification. Rather, those properties are only proven manually – we aim to make sure that they can be proven automatically. We introduce what could be called *selective inlining* – in some cases, as we will show, not all functions should be inlined at the same time, at the risk of losing important information about the terms that we are checking, and they can be inlined in separate, adding essential details to the terms being verified.

Chapter 3

Static Contract Verification

In this section, we will give a detailed explanation of our system, including how a developer's code is processed and how contract definitions are used.

A developer may write code and properties in Haskell. The formal specification for the input language may be found in figure 3.2. This code will be transformed to make clear in what ways it can fail, and with the aid of simplification algorithms and eventually theorem provers – for arithmetic, in particular –, we will check if it is possible that, given the code itself and its properties, we could or not reach an invalid, failing state. Proving that a program cannot reach a failing state is sufficient to prove its safety.

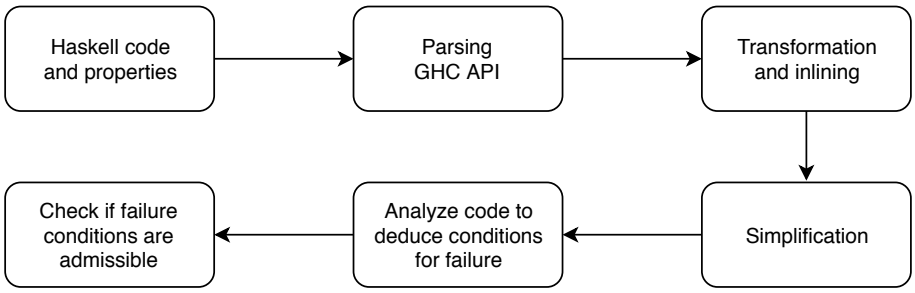


Figure 3.1: The verification module

In order to accept Haskell as input, we need to parse it into a structure over which we can implement the transformation algorithm. We opted to parse Haskell using the GHC API[21], since it allows for access to the same parser as GHC uses, therefore guaranteeing that we get a correct representation of code. The GHC API also enables us to get information about types and returns a representation of optimized and desugared code. This code is represented as Core, an intermediate processing language used by GHC and whose expressions are based on nine constructors, shown in figure 3.3. We manipulate Core into our own language, so that we can apply the transformation and simplification algorithms for checking.

⟨*program*⟩ ::= ⟨*definition*⟩

⟨*definition*⟩ ::= ⟨*contract*⟩ ⟨*function*⟩
| ⟨*function*⟩

⟨*contract*⟩ ::= ⟨*property*⟩
| ⟨*contract*⟩ '->' ⟨*property*⟩

⟨*property*⟩ ::= '{' ⟨*identifier*⟩ '||' ⟨*expression*⟩ '}'

⟨*function*⟩ ::= ⟨*identifier*⟩ '=' ⟨*expression*⟩

⟨*expression*⟩ ::= ⟨*identifier*⟩ ⟨*expression*⟩*
| ⟨*arithmetic, booleans, ...*⟩
| 'case' ⟨*expression*⟩ 'of' ⟨*alt*⟩+

⟨*alt*⟩ ::= ⟨*pat*⟩ '->' ⟨*expression*⟩

⟨*pat*⟩ ::= ⟨*constructor*⟩ ⟨*expression*⟩*
| '_'

Figure 3.2: Formal specification of the input language

```
type CoreExpr = Expr Var

data Expr b = Var Id -- "b" for the type of binders,
  | Lit   Literal
  | App   (Expr b) (Arg b)
  | Lam   b (Expr b)
  | Let   (Bind b) (Expr b)
  | Case  (Expr b) b Type [Alt b]
  | Cast  (Expr b) Coercion
  | Tick  (Tickish Id) (Expr b)
  | Type  Type

type Arg b = Expr b
type Alt b = (AltCon, [b], Expr b)

data AltCon = DataAlt DataCon | LitAlt  Literal | DEFAULT

data Bind b = NonRec b (Expr b) | Rec [(b, (Expr b))]
```

Figure 3.3: Constructors for the Core language

Example 3.1. The following example shows Haskell code for the factorial function, followed by its representation in Core.

```
module Factorial where

fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n-1)
```

```
[$trModule :: Module
 [LclIdX]
 $trModule = Module (TrNameS "main") (TrNameS "Factorial"),
 fact [Occ=LoopBreaker] :: Int -> Int
 [LclIdX]
 fact
   = \ (ds_dIfO :: Int) ->
       case ds_dIfO of { I# ds_dIfQ ->
       case ds_dIfQ of {
         __DEFAULT ->
           * @ Int $fNumInt ds_dIfO (fact (- @ Int $fNumInt ds_dIfO (I#
               1#)));
         0 -> I# 1#
       }
       };]
```

Given that Core is an explicitly-typed language, when analyzing it, we will find type annotations for all variables that are instantiated by the compiler. A module definition is represented by a constructor which takes a package name and the name of the module itself.

During compilation, GHC may perform inlining for code optimization – for example, typeclasses such as *Show* are transformed into dictionary-passing style –, however this could lead to diverging when we have recursive functions. For this reason, the $fact$ function is tagged with a *LoopBreaker*, meaning that the compiler will not inline its definition, at risk of looping. Since the function has two equations, it is made into a case expression, with the compiler performing various sanity checks during this procedure, such as verifying if the cases do not overlap or if there are missing patterns. This case expression is inside a lambda expression, which takes as many arguments as the original function uses in its own definition – any unused arguments are removed during optimization. Variables are instantiated with a random *Unique* code which serves as their key for comparison. This helps in avoiding variable capture, which can occur when we have two variables with the same identifier.

Another step taken by the compiler in processing this function to Core is handling primitive types. *Int* is a primitive type, so it is explicitly represented as such by the $I\#$ constructor – any value with suffix $\#$ is a primitive value. The $fact$ function, as defined in Core, will require that its argument is an Int, and will extract it from its $I\#$ constructor before doing anything with it.

In the recursive case, more instances of explicit typing occur. With *@ Int*, the minus and times operators – which are polymorphic – are specialized to work with *Int*. Their definition for *Int* comes from the *Num* typeclass's dictionary, represented by *$fNumInt*.

3.1 Internal Language

3.1.1 Program language

After we have processed Haskell code into Core, we transform it into our own representation. Its formal specification may be found in figure 3.4. This is important so that we can distinguish relevant expressions, such as primitive operations, the usage of data constructors and function calls. Our representation is aided by certain information supplied by GHC, such as *Name*, which carries an unique identifier given by the compiler, or *AltCon*, which has information about a *Case* alternative, specifying whether it represents a data constructor, for example.

⟨*program*⟩ ::= ⟨*decl*⟩+

⟨*decl*⟩ ::= ⟨*contract*⟩ ⟨*function*⟩

⟨*contract*⟩ ::= ⟨*exp*⟩

⟨*function*⟩ ::= ⟨*id*⟩ '=' ⟨*exp*⟩

⟨*exp*⟩ ::= ⟨*id*⟩
| ⟨*literal*⟩
| 'λ' ⟨*id*⟩ ⟨*exp*⟩
| ⟨*op*⟩ ⟨*exp*⟩+
| ⟨*constructor*⟩ ⟨*exp*⟩*
| ⟨*exp*⟩ ⟨*exp*⟩
| '`case`' ⟨*exp*⟩ '`of`' ⟨*alt*⟩+
| '`let`' ⟨*id*⟩ '=' ⟨*exp*⟩ '`in`' ⟨*exp*⟩
| ⟨*exp*⟩ '◁' ⟨*exp*⟩
| ⟨*exp*⟩ '▷' ⟨*exp*⟩
| '`BAD`'
| '`UNR`'

Figure 3.4: Specification for the intermediate language

```
data ADecl = ADef Name AExp
   (Maybe AExp)

data AExp
 = AVar Name
 | AGlobal Id
 | ALit Literal
 | AApp AExp AExp
 | ALam Name AExp
 | APrimOp PrimOp [AExp]
 | AConApp DataCon [AExp]
 | ACase AExp Name [AAlt]
 | ALet Name AExp AExp
 | ARequires Name AExp AExp
 | AEnsures AExp AExp
 | AUnr Blame
 | ABad Blame

type AAlt = (AltCon, [Name], AExp)

data Blame = Me | CallOf Name
```

Figure 3.5: Constructors for our representation of Haskell code

An *ADecl* represents a top-level declaration for a function, carrying its identifier, the corresponding expression and possibly a contract. An *AExp* represents an expression, with the obvious constructors for the most common expressions, but also support for expressing

conditions, with the *ARequires* and *AEnsures* constructors, or for attributing blame in case of a failure. Blame is attributed with the *AUnr* and *ABad* constructors, which represent either unreachable code (due to a precondition violation) or bad code (a failure due to a postcondition violation), respectively. These allow us to express whether a function should return a value according to its properties, or whether it should be given an argument that respects a given property. This strategy is explained in more detail in the next section, being formalized in definition 3.1. The constructors for this language are detailed in figure 3.5.

This transformation to the *ADecl* representation is done twice – the first time allows us to get the *Name* from each function, which is important for contract parsing, and the second time actually binds each contract to its function. We need the *Name* constructors in order to recognize when a function call occurs in contract definitions. These constructors are supplied by GHC directly and cannot be manually generated, since each *Name* carries an unique identifier as previously described.

The most challenging part with transforming into the intermediate representation is being aware of the fact that a Core *Var* does not always represent a variable *stricto sensu* – the identifier associated with each *Var* may represent one of the following:

- A data constructor
- A primary operation – arithmetic and boolean operators
- A global variable – function call
- A local variable

This information is essential to correctly transform expressions to their adequate *AExp* constructor, making it easier, but not absolutely necessary, to implement simplification rules.

3.1.2 Property language

We support contracts with properties written in Haskell. This is useful for the developer, since it does not require learning a new language and it allows for the definition of more complex properties, through usage of functions directly in the contract definition. While we allow for contracts to be expressed directly in Haskell, with intention of supporting verification for the entire language, it is possible that, due to the inherent nature of static verification, we cannot verify more complex code. We do not have a formal specification of the limits of our system, however this could be an interesting exercise for future work.

```
inc :: a:{x | True} -> {r | r > a}
```

Figure 3.6: A contract for a function that increments a number

```
data HsContract
  = HsParContract HsContract
  -- (contract)
  | HsBaseContract String String
  -- {x | x > 0}
  | HsFunContract (Maybe String) HsContract HsContract
  -- x:{x | x > 0} -> {y | y = x + y}

data Contract = C String HsContract
  -- Function name & Contract
```

Figure 3.7: Constructors for contract definitions

Contracts can be divided into two categories: *flat contracts* and *function contracts*. Flat contracts incide over an argument or value, whereas function contracts are composed of several flat contracts, acting over a function's arguments and over the value returned by it, with the possibility of expressing a postcondition based on the function arguments.

Parsing contracts is done in two steps – firstly, we identify whether their definitions relate to flat or function contracts, and then we work on the expressions themselves, taking care with binding new variables and with function calls. To handle the first step, we devised a Parsec [22] specification for it, which can be found in Appendix A.

For the second step, expressions are parsed using the *haskell-src-meta* package[1]. This package parses expressions into Template Haskell [20] format. Template Haskell (TH) is an extension to Haskell that allows access to the code's abstract syntax tree, as well as to modify it on-the-fly.

```
data Exp
  = VarE Name
  -- ^ @{ x }@
  | ConE Name
  -- ^ @data T1 = C1 t1 t2; p = {C1} e1 e2  @
  | LitE Lit
  -- ^ @{ 5 or \'c\'}@
  | AppE Exp Exp
  -- ^ @{ f x }@
```

Figure 3.8: Template Haskell constructors which we use

[1]This package is available at https://hackage.haskell.org/package/haskell-src-meta

Template Haskell The reader is referred to [20] for more details about Template Haskell, however we give an illustrating example in figure 3.9. The *tuple* function will take a list of any length and build a tuple from it. The use of Template Haskell makes it possible that we define such a function only once for lists of any size, without any type constraints – type checking will only occur when the Template Haskell code is injected.

```
tuple :: ExpQ
tuple = [|\list -> $(tupE (exprs [|list|])) |]
  where
    exprs list = [infixE (Just (list))
                         (varE "!!")
                         (Just (litE $ integerL (toInteger num)))
                  | num <- [0..((length list) - 1)]]

example_usage = $(tuple) [1,2,3,4] -- returns (1,2,3,4)
```

Figure 3.9: A function which turns a list into a tuple of any length

The use of Oxford brackets delimits an expression which we want to be compiled, resulting in an abstract syntax tree which can be inserted into code at runtime. In our case, we have a lambda expression that takes an argument and returns a *splicing* of *tupE* over $exprs[|list|]$. A splice is the evaluation and injection of abstract code – it is the opposite of using Oxford brackets. The *tupE* function returns a constructor which represents a tuple. We can use the *tuple* function by splicing it – it is of type *ExpQ*, which represents an abstract syntax tree for an expression.

Back to our transformation module, we then manipulate the contract expressions into Core format, to reuse the module which converts Core to *ADecl*. It may seem redundant to convert the expressions to Core and only then to our intermediate representation, rather then doing so directly, but this will allow us to centralize the transformation module of our program, ensuring that we have less possible failure points in our implementation. It was also not possible to directly parse the expressions into Core, since the GHC API only supports such parsing during actual compilation.

During conversion to Core, we take attention to whether variables represent function calls (or primary operations) or not. This is important in order to create the adequate Core expressions – as previously mentioned, there are different constructors for either case. Contract definitions are also made into lambda expressions, like what is shown in definition 3.1. Thus, after their conversion into the intermediate representation, we only need to apply functions to their respective contracts in order to obtain a term that we can check for safety. When these lambda expressions are created, we take care to always create new variables with different identifiers, therefore avoiding possible variable capturing during simplification.

To verify a function with its contract, we specify that we would like the function to *ensure*

the contract – this is represented with the notation $e \triangleright t$. Ensuring a contract means that the expression e crashes if it does not respect t, or that it loops if the context does not respect the contract. The context can also be verified, meaning that we want the function to *require* its contract – $e \triangleleft t$. This term will crash if the context does not satisfy the contract t, i.e. an invalid argument may be supplied. These terms, based on application of a function to contracts, are automatically built during program transformation, even considering cases in which a contracted function is called inside another function (which may not be contracted).

```
module Inc2 where

inc :: a:{x | x > 0} -> {r | r > a}
inc :: Int -> Int
inc var = var + 1

---->

[$trModule
 =
 ((TrNameS "main") Module (TrNameS "Inc2"))
 |>
 Nothing,
 inc
 =
 \var. (+ var) 1
 |>
 Just \e. \b. \u. \v_4GG .
  let a_4GI = (\e_4Gm. \b_4Go. \u_4Gq. case (\x_4Gk. (x_4Gk > 0)) e_4Gm
      of
                                        False -> b_4Go
                                        True -> e_4Gm) v_4GG u b
  in (\e_4Gu. \b_4Gw. \u_4Gy. case (\r_4Gs. (r_4Gs > a_4GI)) e_4Gu of
                                  False -> b_4Gw
                                  True -> e_4Gu) (e a_4GI) b u]
```

Figure 3.10: The result of parsing a function and its contract

After being translated into Core, we need to translate contracts into an expression that we can apply to what is being checked. We follow the rules in definition 3.1, instantiating new variables with different identifiers and generating expressions according to the notation. The contract expression is then attached to the function definition itself, being applied and simplified during verification, as described in the next section.

3.2 Verification by program transfomation

In a broader view, we intend to verify programs by transforming them into terms that represent exactly what should happen depending on their contracts, and then checking if invalid execution paths are reachable. More specifically, we test the context for inadequacies, as well as check if certain cases like calls to other functions violate their own contracts during evaluation of another function. This is based on the *ensures/requires* notation, which helps us deal with projecting the properties into a verifiable term.

The *ensures/requires* notation is only powerful enough to help us verify a program if we give a comprehensive specification of what it means and how we can use it to handle difficulties inherent to functional program verification, such as testing higher-order functions.

Definition 3.1 (Ensures/Requires Notation)**.**

$$e \triangleright \{\, x \mid p \,\} = \lambda e \,.\, \lambda b \,.\, \lambda u \,.\, case\,(p[e/x])\; of\; \{True \to e\,;\; False \to b\} \tag{3.1}$$

$$e \triangleleft \{\, x \mid p \,\} = \lambda e \,.\, \lambda b \,.\, \lambda u \,.\, case\,(p[e/x])\; of\; \{True \to e\,;\; False \to u\} \tag{3.2}$$

$$e \triangleright x : t_1 \to t_2 = \lambda e \,.\, \lambda b \,.\, \lambda u \,.\, \lambda v \,.\, (e \triangleright t_2)\,((e \triangleleft t_1)\, v\, u\, b)\, b\, u \tag{3.3}$$

$$e \triangleleft x : t_1 \to t_2 = \lambda e \,.\, \lambda b \,.\, \lambda u \,.\, \lambda v \,.\, (e \triangleleft t_2)\,((e \triangleright t_1)\, v\, u\, b)\, b\, u \tag{3.4}$$

The b and u represent BAD or UNR branches – postcondition failure or unreachable branch due to precondition.

After projecting the contracts according to this notation, we can apply them to $ABad$ or $AUnr$ terms, in order to attribute blame to the function itself or to where the function call originated, as well as determine whether the failure has to do with the function definition itself or with the context under which it was called. This notation also allows for the contracts of other functions to propagate accordingly. If we are verifying a function, we want it to *ensure* its contract; whenever we are checking its arguments, we *require* that they respect a given property. Let us take as an example a function that takes another one as an argument, with a contract being enforced for both functions. This will allow us to demonstrate how the system works for higher-order functions.

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

Figure 3.11: A function that takes another one as an argument and their respective contracts

Following from *Definition 3.1*:

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

In the case of *f1*, this becomes:

$$
\begin{aligned}
&\lambda v_1\,.\,case\,(v_1\,1) >= 0\,of \\
&\qquad True \to case\,(v_1\,1) - 1 >= 0\,of \\
&\qquad\qquad\qquad\qquad\qquad\qquad True \to\ (v_1\,1) - 1 \\
&\qquad\qquad\qquad\qquad\qquad\qquad False \to BAD \\
&\qquad False \to UNR
\end{aligned}
$$

It is clear that the BAD branch represents a failed postcondition for *f1*, given that it is its own expression that would violate the property in this case. A failure in the precondition indicates that an inadequate argument has been supplied. The branch representing this situation is deemed UNR, since it is a failure that is related to the context in which *f1* is called, meaning that the blame should lie with its caller, rather than with the function itself.

Chapter 4

Property checking

The terms described in the previous section represent the different possibilities for failure in contract checking. They are built for each function in a given program and are then tested with relation to the reachability of *BAD* branches. If we can prove that *BAD* branches in a function are unreachable, then we have proven that the function is safe – there is no computation path that violates its postcondition. This can also depend on any conditions we infer from contracts to the function itself or to any other function that is called. These conditions will be assumed true or false, according to the path that would lead to a postcondition failure.

4.1 Simplification

Before we can focus on the different possibilities for failure, we must simplify the terms as much as possible, while keeping all information about preconditions, including some that may have propagated from function calls. Making sure we do not lose any logical conditions is essential to conduct a correct proof of what should be assumed under any computation branch, and of whether or not it is possible to reach it.

In terms of implementation, to ensure that we obtain a term which is simplified as much as possible, we apply the simplification module until we get a fixed point – i.e, $simplify(x) = x$. This relates to simplification only and does not include unrolling. Unrolling is implemented as a separate function which occurs a limited number of times, and after each unroll, we simplify terms to a fixed point once again. Once we are done with simplification and unrolling, we inline any function applications that can be solved according to the definition of functions, such as $length\, Nil$ which can be replaced with 0. An example of this is shown in chapter 5. This is what we call *selective inlining*, and it is paramount to our implementation, since it allows for added detail to the specification of the term we are analyzing, as well as focusing on the definition of the programs themselves. In general, it is done whenever a function is called with a constructor as argument.

To ensure we do not miss any important conditions that propagate from other functions,

```
simplifyF :: AExp -> CEnv -> [ADecl] -> (AExp, CEnv)
simplifyF e c d =
  let
    (a,b) = simplify e c d
  in if a == (fst $ simplify a b d) then (a,b) else simplifyF a
     b d

checkAExp :: Name -> [ADecl] -> ContractEnv -> Int -> CEnv ->
   AExp -> AExp
checkAExp _ ds _ 0 assume e =
  let
    k = fst $ simplifyF e assume ds
    n = fst $ simplifyF (unrollFactsF ds k) assume ds
    n' = unrollFactsF ds n
  in
      -- If unrolling complete...
    if n == n' then n
      -- Else, a new fact was introduced w/fact unroll
    else fst $ simplifyF n' assume ds
checkAExp f ds cEnv n assume e =
  let
    (x,ce) = simplifyF e assume ds
    unr = unrollCalls [f] cEnv ds x
  in
    checkAExp f ds cEnv (n-1) ce unr
```

Figure 4.1: Unrolling and simplifying functions a given number of times

```
inc :: a:{x | x > 0} -> {r | r > a}
inc var = var + 1

inc =
 \v_4GG. case (v_4GG > 0) of
          T -> case ((+ v_4GG) 1 > case (v_4GG > 0) of
                                     T -> v_4GG) of
                    F -> BAD Me
                    T -> (+ v_4GG) 1
```

Figure 4.2: The result of applying the previous algorithm to the function in figure 3.10

we need to replace any function calls with the correct terms, according to the ensures/requires notation – this guarantees that they are projected correctly. The initial term is wrapped in an *AEnsures* constructor, with the absence of a contract meaning that it should respect $\lambda\, e\, b\, u.\, e$ – a dummy contract which is only used for consistency of the implementation. Any function calls are represented by the *ARequires* constructor, meaning that any potential violation of those functions' contracts will not be represented as a failure of the initial term itself.

```
inlineContractDef :: ContractEnv -> AExp -> AExp
inlineContractDef cEnv ex =
  case ex of
    AGlobal i -> case (lookupContractEnv cEnv (idName i)) of
      Nothing -> (AGlobal i)
      Just c ->  (ARequires (idName i) (AGlobal i) (toAExp c))
    AVar v -> case (lookupContractEnv cEnv v) of
      Nothing -> AVar v
      Just c ->  (ARequires v (AVar v) (toAExp c))
    ...

checkAndBuild :: ContractEnv -> Int -> [ADecl] -> [ADecl] ->
   [ADecl]
checkAndBuild _ _ _ [] = []
checkAndBuild cEnv n bs ((ADef f e t):ds) =
  case t of
    Nothing ->
      let
        fbody = inlineContractDef cEnv e
        body2 = AEnsures fbody okContract
        a = checkAExp f bs cEnv n [] body2
      in
        (ADef f a (Just okContract)) : checkAndBuild cEnv n bs ds
    Just rt ->
      let
        fbody = inlineContractDef cEnv e
        body2 = AEnsures fbody rt
        a = checkAExp f bs cEnv n [] body2
      in
        (ADef f a t) : checkAndBuild cEnv n bs ds
```

Figure 4.3: Inlining contract definitions for calls to other functions

Terms are reduced according to standard Haskell semantics. We are only dealing with lambda expressions, data constructors and case expressions, thus it is simple enough to explain how

simplification works. Lambda expressions are beta-reduced and case expressions are matched against the correct case, based on the constructor of the scrutinee. When we do not have enough information to match in a case expression (e.g. we are scrutinizing a variable), we leave it as it is, but still simplify its alternatives, considering if we should assume a variable to be of a certain constructor or not – this depends on the branch that we are currently traversing. Other more advanced simplifications for case expressions include checking if all branches are the same and reducing it to that same branch, removing *UNR* branches to shrink the expression or pushing an expression into every alternative when the case expression is being applied. The rules used are formally defined below.

Definition 4.1 (Simplification and transformation rules)**.**

$$
\begin{aligned}
f &\to \lambda x.e \ \ where \ \ \Delta[f \mapsto (\lambda x.e)] \\
(\lambda x.e_1)e_2 &\to e_1[e_2/x] \\
case\, K_i\, y_i\, of\, \{\ldots K_i x_i \to e_i \ldots\} &\to e_i[y_i/x_i] \\
case\, K \ldots\, of\, \{K'_i \ldots \to e_i\,, \ldots, \_ \to e\} &\to e \ \ where \ \ \forall i, K \neq K'_i \\
case\, K \ldots\, of\, \{K'_i \ldots \to e_i\,, \ldots\} &\to UNR \ \ where \ \ \forall i, K \neq K'_i \\
case\, (case\, e\, of\, alts_i)\, of\, alts_j &\to case\, e\, of\, (p_i \to case\, e_i\, of\, alts_j) \\
&\qquad where \ \ (p_i \to e_i) \in alts_i \\
(case\, e\, of\, alts_i)\, a &\to case\, e\, of\, (p_i \to e_i\, a) \\
&\qquad where \ \ (p_i \to e_i) \in alts_i \\
case\, e\, of\, (p_i \to e_i \ldots p_j \to UNR) &\to case\, e\, of\, (p_i \to e_i) \\
case\, e\, of\, (K_i\, x_i \to e_i) &\to case\, e\, of\, (K_i\, x_i \to e_i[K_i\, x_i/e]) \\
case \ \ BAD \ \ of \ldots &\to BAD \\
case \ \ UNR \ \ of \ldots &\to UNR
\end{aligned}
$$

All rules are applied in the same order as they are presented, except for the first rule (looking up a function definition), which is applied only during unrolling. Unrolling consists in inlining a function definition, along with its contract, and we only do so a limited number of times. This helps to deal with the possibility of non-termination.

One particular case in implementing these rules relates to the expression scrutinization rule. We need to keep track of the substitutions we are performing, which will be reused for the expressions in each alternative. This is necessary to keep the substitutions sound. When a scrutinee appears for the first time, we rebuild the case expression as such:

1. Check which constructors are covered by the alternatives
2. Simplify each alternative:

- If our alternative is a *DEFAULT*:
  - If the constructors are literals (e.g. numbers), or if we are missing more than one constructor, we just simplify the right-hand side of the alternative – it means the *DEFAULT* branch is, indeed, needed
  - If there are no missing constructors, the *DEFAULT* clause is redundant and is deemed unreachable (*AUnr*)
  - Otherwise, there is just one constructor missing, and we modify the alternative from *DEFAULT* to it, and simplify the right-hand side
- If our alternative is not a *DEFAULT*, we simplify the right-hand side and add an entry to our environment, indicating which constructor and variable bindings are valid for this branch

For this reason, the simplification function returns a tuple with the reduced expression and with an environment of the choices that have already been made. Let us take the following expression as an example:

```
case v of
  Nil -> 0
  Cons x y -> func v
```

In this case, any other occurence of v in the $Cons\ x\ y$ branch must be substituted by $Cons\ x\ y$, and any other case which scrutinizes v must follow the $Cons\ldots$ branch. All other case simplification rules apply strictly as defined in definition 4.1, except for whenever we have arithmetic formulas – we do not touch these case expressions, to preserve the information that these formulas give us for interfacing with theorem provers. This also allows us to return a more comprehensive specification of what exactly leads to unsafety in a program.

4.2 Interfacing with provers

After applying the contracts to their respective functions and simplifying the terms, we need to analyze the resulting code to check if it is safe or not.

For many cases, it is sufficient to check if any *BAD* branch remains in the code. If it does not, then we can conclude that a function is safe, according to its contract. If it does, we need to check if these *BAD* branches are reachable – in certain cases, preconditions or contracts coming from calls to other functions can make them unreachable. We should, therefore, traverse the case expressions to find the *BAD* terms, and take note of what should be assumed to be true or false, to reduce the expression to *BAD*. Sometimes, multiple failure branches appear – we need to check every single one of them independently.

The *toBAD* function takes a case expression and analyzes it recursively, in order to extract the formulas that are scrutinized in the expression. Once a formula is found, we can assume it to

```
toBAD :: AExp -> [[AExp]]
toBAD tree@(ACase _ _ _) = map reverse $ traverse [] tree
  where
    traverse path (ACase sc v ps) =
      concat $
      map (\(c, _, r) ->
      if c == DataAlt trueDataCon then
        (traverse (sc:path) r) else
          if c == DataAlt falseDataCon then
            (traverse ((APrimOp NotOp [sc]):path) r)
            else (traverse path r)) ps
    traverse path (AApp _ a2) = traverse path a2
    traverse path (APrimOp _ es) = concatMap (traverse path) es
    traverse path (AConApp _ es) = concatMap (traverse path) es
    traverse path (ABad _) = [path]
    traverse _ _ = []
toBAD (ALam x e) = toBAD e
toBAD _ = [[]]
```

Figure 4.4: The function that returns a list of conditions that lead to each failure possibility

be true or false, depending on the branch that we follow next – if, however, the scrutinee is not a formula (e.g. a function call), then we ignore it, but continue following through the various alternatives. Once we reach a *BAD* term, we have a path to failure. This function is applied to all functions that have been declared in the developer's code.

Let us now take the following example to show how a contract is inlined and which conditions we should pass to a prover for checking.

The precondition $x > 0$ is violated if an argument is less or equal to 0. However, we can discard this branch using simplification rules – and for safety checking purposes, we can assume that preconditions will not be violated.

In general, and as previously mentioned, we assume conditions true or false according to what path we are following, since we could need them to help prove the truthness of a postcondition. These assumptions can then be pushed onto the stack of any prover that supports truth stacks, such as Z3 [6] or Simplify [5].

The postcondition $x + 1 > x$ is, obviously, never violated, which shows that this function is safe – the *False* branch is unreachable and should be discarded, leaving the term with no *BAD*. For the purposes of this implementation, we would need to have the prover assume that $x > 0$, since we followed the *True* branch on that condition. This assumption extends to the inner case in the postcondition – $case\, x > 0\, of\, True \rightarrow x$ is reduced to x, according to the conditions we

```
inc :: a:{x | x > 0} -> {r | r > a}
inc x = x + 1

This function becomes:

\x . case x > 0 of
    T -> case x + 1 > (case x > 0 of
                          T -> x
                          F -> UNR) of
           T -> x + 1
           F -> BAD Me
    F -> UNR
```

Figure 4.5: A function, its contract and its inlining

```
(inc, [[(x > 0),
        (not (x + 1) > case (x > 0) of
                          T -> x))]])
```

Figure 4.6: Possibilities for failure of *inc*, as given by *toBAD*

collected in this path. This special reduction ensures that, right before we send any expressions to a theorem prover, we are only left with first-order formulas. In figure 4.7, we may find the main functions for interfacing with some theorem prover.

The *checkProver* function receives a *Prover* (which contains input and output handles to communicate with the theorem prover process), a list of assumptions and a list of expressions which represent the conditions for a failing branch.

Given a condition, we check if it is made of integers only, and verify it immediately and independently of other assumptions – being made solely of integers and boolean operators, it will not depend on any other condition. We then store the result that we read from the theorem prover – valid or invalid – in an environment, which we will use to propagate any intermediate proofs such as this one. The result comes from parsing the output from the prover. (As an example, a parser for *Simplify*'s output may be found in appendix B.) If it is not an integer-only condition, we check if there are any case expressions left over, such as $case\, x > 0\, of\, True \rightarrow x$ in the previous example. We solve these expressions based on assumptions that have already been made, checking the assumption environment for this. These assumptions are propagated across expressions using the *solveCase* function.

Once an expression is only a simple formula, devoid of any case expressions, we push it onto the prover's truth stack. After we are only left with one last expression, we check if it is valid

```
data Prover = P { inH, outH :: Handle }

checkProver :: Prover -> [(AExp, Bool)] -> [AExp] -> IO ()
checkProver p ass [x] = do
  let e = (solveCase ass x)
  send p (formulaToString (toFormula e))
  printLines p
checkProver p ass (x:xs) = do
  case intsOnly x of
    True -> do
      send p (formulaToString (toFormula x))
      val <- findResponse p validOrInvalid
      checkProver p ((x, val):ass) xs
    False -> do
      let iN = isNot x
      let iC = hasCase x
      case iC of
        True -> do
          checkProver p ass ((solveCase ass x) : xs)
        False -> do
          pushProver p x
          checkProver p ((x,not $ iN):ass) xs

solveCase ass (APrimOp o es) = (APrimOp o (propagate ass es))

propagate _ [] = []
propagate ass (e@(APrimOp o es):x) = solveCase ass e : propagate
   ass x
propagate ass ((ACase sc var ps):x) = (case lookup sc ass of
  Just True -> find (DataAlt trueDataCon) ps
  Just False -> find (DataAlt falseDataCon) ps) : propagate ass x
propagate ass (e:es) = e : propagate ass es
```

Figure 4.7: Interfacing with a theorem prover, taking advantage of assumptions

or not, according to the assumptions we have pushed – we are verifying if all these conditions imply that the final one will have a truth value that makes its case expression reduce to *BAD*. If this implication is satisfiable, then *BAD* is reachable, otherwise, *BAD* is unreachable, since the assumptions are incompatible with a postcondition failure, and it can be discarded. If we prove that all implications are unsatisfiable, then the function is safe.

In order to send expressions to a prover, they need to be converted to SMT-LIB syntax, which is the input language used by Z3 and Simplify. The SMT-LIB Standard [23] has been defined as the most common language for an SMT solver to take as input, having been created with the help of the developers of theorem provers. In this language, arithmetic and relational formulas are always presented in prefix notation – for example, $1 + len\, xs \rightarrow (+\, 1\, (len\, xs))$.

```
~$ z3 -in
  (declare-const x Int)
  (assert (> x 0))
  (assert (not (> (+ x 1) x)))
  (check-sat)

  unsat
```

Figure 4.8: Using Z3 to prove that *BAD* is not reachable

Thus, our system transforms code in order to make it clearer how it could fail, and analyzes the output of that transformation in order to check how the code itself or its preconditions can imply failure. Finally, it checks if the conditions that lead to errors are, indeed, admissible.

Chapter 5

Working Examples

In this section, we give a better explanation of what our system can do in terms of verification, starting with smaller programs over simple arithmetic and finishing with programs that have recursion over data structures.

Example 5.1 (Analyzing function calls and detecting invalid contexts)**.** Let us take an example to illustrate how an inadequate context is detected by this system. Taking the function in figure 4.2, let us assume we have another function that simply calls *inc* with argument 1 – i.e. $f = inc\,1$. In our system, this call would be transformed and inlined as follows:

```
case (1 > 0) of
 F -> BAD Me
 T -> case ((inc 1) > case (1 > 0) of
                        F -> BAD Me
                        T -> 1) of
       T -> case (1 > 0) of
              F -> BAD Me
              T -> case (((+ 1) 1) > case (1 > 0) of
                                       F -> BAD Me
                                       T -> 1) of
                     T -> (+ 1) 1
```

Figure 5.1: Checking the call *inc* 1

As far as the system is concerned, when checking f, we will only be focusing on it potentially violating *inc*'s precondition – after all, it is the only violation it could make, given that it has no contract, but it is calling a function that has one. This means that the only possible failure would have to do with 1 being lesser than 0, violating the precondition $x > 0$. A theorem prover concludes that this is false, and therefore the BAD branches are unreachable, proving that this call to *inc* is safe.

Example 5.2 (Ensuring a function will not fail with preconditions)**.** To show how important preconditions are, we can also take as an example the verification of the *head* function for lists. This function is undefined for empty lists, since we have no element to return, and a call to *head* with an empty list will throw an exception. Therefore, an adequate precondition is ensuring that the list is not empty.

```
data List a = Cons a (List a) | Nil

not' :: Bool -> Bool
not' False = True
not' True = False

null' :: List a -> Bool
null' Nil = True
null' (Cons _ _) = False

head' :: a:{x | not' (null' x)} -> {r | True}
head' :: List Int -> Int
head' (Cons n _) = n
```

Figure 5.2: Verifying the head function for lists

Applying the contract to the function and inlining function calls, we obtain the following term:

```
\v. case (\ds_D. case ds_D of
                 F -> T
                 T -> F) ((\ds_t. case ds_t of
                                  Cons _ _ -> F
                                  Nil -> T) v) of
    F -> (\ds_l. case ds_l of
                  Nil -> patError
                  Cons n _ -> n) (UNR Me)
    T -> (\ds_l. case ds_l of
                  Nil -> patError
                  Cons n l -> n) v
```

Figure 5.3: Paths for verification of *head*

Applying simplification rules, we obtain the following case expression:

```
\v. case v of
     Cons n l -> n
     Nil -> UNR Me
```

Figure 5.4: Simplified version of *head*

Under the simplification rules for cases, we can discard the *Nil* branch. Therefore, the precondition is strong enough to ensure we do not reach an error branch, and the function is safe. Any wrongful call to it will also be caught by this system, similarly to the first example.

Example 5.3 (Checking a recursive arithmetic function)**.** Recursion is at the heart of nearly every functional program, being essential to express certain concepts clearly. One example of this is with the factorial function, which is only defined for numbers greater or equal to zero. Our system can handle this kind of functions easily. The definition of $fact$ and its contract follows below.

```
fact :: a:{x | x >= 0} -> {r | r >= 1}
fact :: Int -> Int
fact x = case x == 0 of
  T -> 1
  F -> case x > 0 of
   T -> x * fact (x-1)
   F -> error "Invalid argument"
```

Figure 5.5: The factorial function and its contract

It is important to mention that the *error* call is made into a *BAD* term. Other than that, simplification follows as usual, producing the failing branches listed in figure 5.6.

```
[(v_4GG >= 0), (not (v_4GG == 0)),
 (not (v_4GG >= 0))]

[(v_4GG >= 0), (not (v_4GG == 0)),
 (v_4GG > 0),
 (not (v_4GG - 1 >= 0))]

[(v_4GG >= 0), (not (v_4GG == 0)),
 (v_4GG > 0), (v_4GG - 1 >= 0),
 (fact (v_4GG - 1) >= 1),
 (not (v_4GG * (fact (v_4GG - 1)) >= 1))]

[(v_4GG >= 0), (v_4GG == 0),
 (not (1 >= 1))]
```

Figure 5.6: Possibilities of failure for $fact$

The first possibility is for whenever a top-level call to $fact$ happens with a number which is strictly lesser than zero. This violates $fact$'s precondition. The second possibility relates to a case in which the recursive call could violate $fact$'s precondition, if $x - 1 < 0$ – the assumptions, however, make this case unreachable as well. The third possibility assumes that $fact(x-1) >= 1$ and would fail if $x * fact(x-1) < 1$ – violating $fact$'s postcondition, which is proven impossible. The recursive call is, at this point, proven to be correctly defined according to the function's specification. The final possibility would fail if $1 < 1$, which is obviously false, thus proving that $fact$'s base case conforms to its contract and that the function is safe.

Example 5.4 (Verifying programs that work over lists)**.** This system can also deal with more advanced programs, including those with recursion. As an example, let us consider the *append* function, which takes two lists and returns their concatenation. One condition that should be verified relates to the length of the resulting list – it should be the same as the sum of the lengths of the input lists.

```
append' :: a:{x | True} -> b:{y | True} -> {r | len r == len a +
   len b}
append' :: List Int -> List Int -> List Int
append' Nil ys = ys
append' (Cons x xs) ys = Cons x (append' xs ys)
```

Figure 5.7: Defining *append* and its contract

After applying the contract to the function, we get two BAD branches – one for each definition of *append*, considering whether the first list is empty or not.

```
\v1 v2. case v1 of
        Cons x xs -> case (len (Cons x (case (len (append' xs v2) == (len xs + len v2))
                                             T -> append' xs v2)) == (len (Cons x xs) + len v2))
                       F -> BAD Me
                       T -> Cons x (case (len (append' xs v2) == (len xs + len v2))
                                         T -> append' xs v2)
        Nil -> case (len v2 == (len (Nil) + len v2))
                 F -> BAD Me
                 T -> Cons x (case (len (append' xs v2) == (len xs + len v2))
                                   T -> append' xs v2)
```

Figure 5.8: *append*'s contract helps understand how it could fail

The system will then apply the definition of *len* to the calls $len\,Nil$ and $len\,(Cons\ldots)$, giving us something much more expressive to work with. This is an example of the *selective inlining* mentioned in section 2.3.

```
\v1 v2. case v1
        Cons x xs -> case (len (append' xs v2) == (len xs + len v2))
                       T -> case (1 + len (append' xs v2) == (1 + len xs + len v2))
                              F -> BAD Me
                              T -> Cons x (case (len (append' xs v2) == (len xs + len v2))
                                                T -> append' xs v2)
        Nil -> case (len v2 == 0 + len v2))
            F -> BAD Me
                 T -> Cons x (case (len (append' xs v2) == (len xs + len v2))
                                   T -> append' xs v2)
```

Figure 5.9: The definition of *len* is essential to prove the safety of *append*

In order to prove that *append* is correctly defined for the case in which the first list is empty, it suffices to show that $len\,v2 = 0 + len\,v2$, as expected. To show that *append* is properly defined for the case where the first list is not empty, we need to assume that its postcondition $len\,(append'\,xs\,v2) == (len\,xs + len\,v2)$ will hold for the recursive case, as described in [7]. After one step of execution of *append*, the result will be $Cons\,x\,(append'\,xs\,v2)$, which has length $1 + len\,(append'\,xs\,v2)$. We want this length to be equal to $1 + len\,xs + len\,v2$, due to *append*'s precondition. This is proven true due to the assumption we made, proving both BAD branches are unreachable and that *append* is safely defined.

The *append* function is the backbone of most functions that work over lists. We can use it to define other predicates that perform certain operations with them, including *reverse*, which returns a reversed list. We will impose that the returned list must have the same length as the input list.

```
reverse' :: a:{x | True} -> {r | len' r == len' a}
reverse' :: List Int -> List Int
reverse' Nil = Nil
reverse' (Cons x xs) = append' (reverse' xs) (Cons x Nil)
```

Figure 5.10: Defining *reverse*, with the help of *append*, and its contract

We then apply the contract to the function, leaving us with two possibilities for failure, one for each definition of the function. The conditions for either possibility are shown below.

```
[(not (0 == 0))]

[(len (reverse' xs) == len xs),
 (len (append' (reverse' xs) (Cons x Nil)) == len (reverse' xs)
    + 1 + len Nil),
 (not (len (append' (reverse' xs) (Cons x Nil))) == 1 + (len
    xs))]
```

Figure 5.11: The possibilities for *reverse* to fail

The first case refers to when we are trying to reverse an empty list. The function would fail if its returned list had a length different than the length of the first list, meaning that, in this case, it would fail if 0 were different from 0. This is deduced from the definition of $reverse\,Nil$. This, obviously, does not hold, so *reverse* is safe for its base case.

The second case, which uses recursion, has two conditions that hold before we can look at a possible postcondition failure. The first condition refers to the inductive property of the postcondition – we should assume that it holds for the recursive case. The second condition is derived from *append*'s contract – we can assume its postcondition to hold, since we are not concerned with verifying *append* itself at this point. The final condition is *reverse*'s postcondition, and for *reverse* to be deemed unsafe, it cannot hold. However, based on the other two conditions, it is impossible for *reverse*'s postcondition to be false, as a theorem prover shows. *reverse* is proven safe, with the help of conditions derived from the functions we called.

Example 5.5 (Verifying programs that work over arbitrary data structures)**.** Our implementation is capable of handling programs with arbitrary data structures, such as trees, and prove properties over them. As an example, let us take a function that gives us the depth of a tree.

```
data Tree a = Node a (Tree a) (Tree a)
  | Leaf

depth :: a:{x | True} -> {r | r >= 0}
depth :: Tree Int -> Int
depth Leaf = 0
depth (Node _ a b) = 1 + (if (depth a) > (depth b) then depth a
   else depth b)
```

Figure 5.12: Defining *depth* and its contract

Applying the contract to the function definition, we get a term with three possibilities for failure – one for when we are taking the depth of a tree which just has a leaf, and two depending on which branch goes deeper.

```
\v. case v of
      Node _ a b ->
       case (depth a >= 0) of
        T -> case (depth b >= 0) of
              T -> case depth a > depth b of
                    F -> case (depth b >= 0) of
                          T -> case (1 + depth b >= 0) of
                                F -> BAD Me
                                T -> case (depth a >= 0) of
                                      T -> case (depth b >= 0) of
                                            T -> case (depth b >= 0) of
                                                  T -> 1 + depth b
                    T -> case (depth a >= 0) of
                          T -> case (1 + depth a >= 0) of
                                F -> BAD Me
                                T -> case (depth a >= 0) of
                                      T -> case (depth b >= 0) of
                                            T -> case (depth b >= 0) of
                                                  T -> 1 + depth b
      Leaf ->
       case (0 >= 0) of
        F -> BAD Me
        T -> case (depth a >= 0) of
              T -> case (depth b >= 0) of
                    T -> case depth a > depth b of
                          F -> case (depth b >= 0) of
                                T -> 1 + depth b
                          T -> case (depth a >= 0) of
                                T -> 1 + depth a
```

Figure 5.13: Applying *depth*'s contract

Analyzing this term according to our strategy, we extract the conditions that lead to each possible failure.

```
[(depth a >= 0), (depth b >= 0),
((depth a) > (depth b)), (depth a >= 0),
(not (1 + (depth a) >= 0))]

[(depth a >= 0), (depth b >= 0),
(not ((depth a) > (depth b))), (depth b >= 0),
(not (1 + (depth b) >= 0))]

[(not (0 >= 0))]
```

Figure 5.14: Possibilities for *depth* to fail

The first case relates to when the left branch goes deeper – $depth\, a > depth\, b$. We assume either branch to have $depth >= 0$ – it is our inductive property – and this case would only fail if $depth\, a + 1 <= 0$, which is obviously false, given our assumptions. The second case is analogous to the first, but with the right branch going deeper. The last case refers to when we are taking the depth of a tree with just a leaf, and it is clear that this case respects the specification for the *depth* function.

Example 5.6 (Recursion with two different data structures and multiple function calls)**.** Let us now verify a function that converts between two different data structures. We will convert a tree to its flattened representation (a list) and check if we have the same number of nodes as elements in the new list. This means that we will also have to define a function for counting nodes, and its respective contract. Both definitions can be found below.

```
collapse :: a:{x | True} -> {r | len r == count a}
collapse :: Tree Int -> List Int
collapse Leaf = Nil
collapse (Node n a b) = append' (append' (collapse a) (Cons n
   Nil)) (collapse b)

count :: a:{x | True} -> {r | r >= 0}
count :: Tree Int -> Int
count Leaf = 0
count (Node _ a b) = 1 + count a + count b
```

Figure 5.15: Definition of *collapse* and *count* and their contracts

Verifying *count* follows analogously to the verification of *depth*. The verification of *collapse* is,

however, slightly more complex – it depends on the definitions of three other functions: *append*, *length* and *count*. This is evidenced by the conditions that we infer from applying *collapse*'s contract to itself.

```
[(not (0 == 0))]

[(len (collapse a) == count a),
 (len (append' (collapse a) (Cons n Nil)) == (len (collapse a) +
    1 + 0)),
 (len (collapse b) == count b),
 (len (append' (append' (collapse a) (Cons n Nil)) (collapse b))
    == (len (append' (collapse a) (Cons n Nil)) + len (collapse
    b))),
 (not (len (append' (append' (collapse a) (Cons n Nil))
    (collapse b)) == (1 + (count a) (count b)))]
```

Figure 5.16: The cases in which *collapse* could fail

The base case is proven directly. For the inductive case, we need to assume four conditions:

- The length of collapsing the left branch is the same as the left branch's node count
- The length of collapsing the right branch is the same as the right branch's node count
- The length of appending the left list to a list with the current node's value equals the length of the left list plus one (follows from the definitions of *append* and *length*)
- The length of appending the left list and the current node to the right list equals the length of the left list and the current node plus the length of the right list (follows from the definitions of *append* and *length*)

Given these conditions, *collapse* would only fail if the length of the lists on the left and right sides, plus the current node, were different from the node counts on both sides plus one (the current node). This is not possible due to the assumptions we made, so *collapse* is proven safe.

Profiling Below, we show some statistics about verifying the aforementioned examples, including time, number of verification conditions generated and memory usage. We grouped the example functions into four modules, as shown below.

The modules shown in table 5.1 contain the following functions and contracts:

- Inc (*inc*):

```
inc :: a:{x | True} -> {r | r > a}
```

- Fact (*fact*):

```
fact :: a:{x | x >= 0} -> {r | r >= 1}
```

- Cons (*head*, *length*, *append*, *reverse*, *map*):

```
head' :: a:{x | not' (null' x)} -> {r | True}
length' :: a:{x | True} -> {r | r >= 0}
append' :: a:{x | True} -> b:{y | True} -> {r | length' r ==
   (length' a + length' b)}
reverse' :: a:{x | True} -> {r | length' r == length' a}
map' :: a:{x | True} -> l:{y | True} -> {r | length' r ==
   length' l}
```

- Tree (*depth*, *count*, *mapTree*, *collapse*, *length*, *append*):

```
depth :: a:{x | True} -> {r | r >= 0}
count :: a:{x | True} -> {r | r >= 0}
mapTree :: a:{x | True} -> b:{y | True} -> {r | count r ==
   count b}
collapse :: a:{x | True} -> {r | length' r == count a}
length' :: a:{x | True} -> {r | r >= 0}
append' :: a:{x | True} -> b:{y | True} -> {r | length' r ==
   (length' a + length' b)}
```

Module	# Functions	Time (s)	Memory (kB)	# VCs
Inc	1	3,20	33,9	2
Fact	1	12,15	48,8	16
Cons	5	24,18	42,5	13
Tree	6	39,37	59,2	31

Table 5.1: Time, memory and number of verification conditions generated for each module

Analyzing the results, we note that time is not an issue for these examples, being verified quickly enough. It is also noticeable that the number of generated verification conditions is not linear to the number of functions in each module, and can indeed be much larger when we have functions that return recursive function calls, like $fact$. Finally, we realized that memory usage escalates with the number of verification conditions that are generated.

Chapter 6

Conclusion and Future Work

Static contract checking is an important concept in the area of software verification, since it allows checking code for safety without running it, therefore allowing a programmer to find defects earlier in development. It is also important for functional languages, even considering that their use often results in safer, better designed code. It could also be argued that it actually becomes even more important when we are dealing with functional programming, in spite of a tendency to produce more correct code, since languages like Haskell allow for easier checking of more complex properties over software, given their declarative properties. In this dissertation, we focused on a theoretical model for static contract checking and implemented it in practice, with clear, reasonable examples of how useful it can be to ensure that code is adequately written. Contracts also force the developer to think in a different way: they lead to a greater focus on what properties should code have. This implementation extends the one described in the original work, enhancing support for recursion and arbitrary data structures, making it more suitable for real-life programs.

6.1 Future Work

Future work includes interfacing with interactive theorem provers such as Coq, and implementing a dynamic contract checking module, thus creating a hybrid contract checking system with greater coverage. As evidenced in chapter 1, dynamic verification has advantages over static verification, particularly due to the fact that we can test programs against real execution traces, and a hybrid system would be able to take advantage of each approach's positives.

6.2 Final Remarks

This is a step forward in functional program verification, since the produced work is capable of detecting faults in a very expressive subset of Haskell and it can easily be adapted to verify other functional programming languages. The main difficulties with this implementation were handling

certain details in the Core specification, as well as dealing with the possibility of non-termination. At the end of the work described in this thesis, we think that the result is a step forward in contract verification, which is important on its own, given how it helps developers write better, more correct code. Given Haskell's declarative style, it is also a perfect fit for the development of tools that parse and analyze code, and that has been evidenced throughout this dissertation.

Bibliography

[1] D. S. Rosenblum. "A practical approach to programming with assertions". In: *IEEE Transactions on Software Engineering* 21.1 (Jan. 1995), pp. 19–31. ISSN: 0098-5589. DOI: 10.1109/32.341844.

[2] Ralf Hinze, Johan Jeuring and Andres Löh. "Typed contracts for functional programming". In: *Proceedings of FLOPS 2006: Eighth International Symposium on Functional and Logic Programming*. Vol. 6. Springer. 2006, pp. 208–225.

[3] Dana Xu. "Static Contract Checking for Haskell". PhD thesis. University of Cambridge Computer Laboratory, 2008.

[4] Dana Xu, Simon Peyton Jones and Koen Claessen. "Static Contract Checking for Haskell". In: *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '09. Savannah, GA, USA: ACM, 2009, pp. 41–52. ISBN: 978-1-60558-379-2. DOI: 10.1145/1480881.1480889. URL: http://doi.acm.org/10.1145/1480881.1480889.

[5] David Detlefs, Greg Nelson and James B Saxe. "Simplify: a theorem prover for program checking". In: *Journal of the ACM (JACM)* 52.3 (2005), pp. 365–473.

[6] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[7] Simon Thompson. *Haskell: the craft of functional programming*. Vol. 2. Addison-Wesley, 2011.

[8] Rui Andrade and Mário Florido. "Contract Checking for Lazy Functional Languages". In: *Proceedings of INForum 2018 - Simpósio de Informática*. Coimbra, Portugal, 2018.

[9] Luis Damas and Robin Milner. "Principal Type-schemes for Functional Programs". In: *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '82. Albuquerque, New Mexico: ACM, 1982, pp. 207–212. ISBN: 0-89791-065-6. DOI: 10.1145/582153.582176. URL: http://doi.acm.org/10.1145/582153.582176.

[10] Hongwei Xi and Frank Pfenning. "Dependent types in practical programming". In: *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1999, pp. 214–227.

[11] Edwin C Brady. "IDRIS—: systems programming meets full dependent types". In: *Proceedings of the 5th ACM workshop on Programming languages meets program verification*. ACM. 2011, pp. 43–54.

[12] Ana Bove, Peter Dybjer and Ulf Norell. "A brief overview of Agda–a functional language with dependent types". In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 2009, pp. 73–78.

[13] Adam Chlipala. *Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant*. MIT Press, 2013.

[14] Lawrence C Paulson. *Isabelle: A generic theorem prover*. Vol. 828. Springer Science & Business Media, 1994.

[15] W Richards Adrion, Martha A Branstad and John C Cherniavsky. "Validation, verification, and testing of computer software". In: *ACM Computing Surveys (CSUR)* 14.2 (1982), p. 165.

[16] Niki Vazou et al. "Refinement types for Haskell". In: *ACM SIGPLAN Notices*. Vol. 49. 9. ACM. 2014, pp. 269–282.

[17] Leonardo De Moura and Nikolaj Bjørner. "Satisfiability Modulo Theories: Introduction and Applications". In: *Commun. ACM* 54.9 (Sept. 2011), pp. 69–77. ISSN: 0001-0782. DOI: 10.1145/1995376.1995394. URL: http://doi.acm.org/10.1145/1995376.1995394.

[18] Niki Vazou, Eric L Seidel and Ranjit Jhala. "Liquidhaskell: Experience with refinement types in the real world". In: *ACM SIGPLAN Notices*. Vol. 49. 12. ACM. 2014, pp. 39–51.

[19] Robert Bruce Findler and Matthias Felleisen. "Contracts for higher-order functions". In: *ACM SIGPLAN Notices*. Vol. 37. 9. ACM. 2002, pp. 48–59.

[20] Tim Sheard and Simon Peyton Jones. "Template meta-programming for Haskell". In: Oct. 2002, pp. 1–16. URL: https://www.microsoft.com/en-us/research/publication/template-meta-programming-for-haskell/.

[21] Cordelia Hall et al. "The Glasgow Haskell Compiler: A Retrospective". In: *Functional Programming, Glasgow 1992*. London: Springer London, 1993, pp. 62–71. ISBN: 978-1-4471-3215-8.

[22] Daan Leijen and Erik Meijer. *Parsec: Direct style monadic parser combinators for the real world*. Tech. rep. 2001.

[23] Clark Barrett, Aaron Stump, Cesare Tinelli et al. "The SMT-LIB Standard: Version 2.0". In: *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*. Vol. 13. 2010, p. 14.

Appendix A

Parser for contracts

```
type P = Parser

lexer = Token.makeTokenParser emptyDef
identifier = Token.identifier lexer
whiteSpace = Token.whiteSpace lexer

contracts :: P [Contract]
contracts = many $ do
  fname <- identifier
  whiteSpace
  string "::"
  whiteSpace
  cont <- hsContract
  whiteSpace
  return $ C fname cont

hsContract :: P HsContract
hsContract =  (try funContract) <|> baseContract

baseContract :: P HsContract
baseContract = try (do
  char '{'
  whiteSpace
  var <- identifier
  whiteSpace
  exp <- between (char '|') (char '}') (many1 $ noneOf ("}"))
  return $ HsBaseContract var exp) <|> parContract
```

```
parContract :: P HsContract
parContract = do
  char '('
  whiteSpace
  h <- hsContract
  whiteSpace
  char ')'
  return $ HsParContract h

funContract :: P HsContract
funContract = do
  n <- optionMaybe (do {var <- identifier; char ':'; return var})
  l <- baseContract
  whiteSpace
  string "->"
  whiteSpace
  r <- hsContract
  return $ HsFunContract n l r

unVar (HsBaseContract v _) = v

parseString :: String -> [Contract]
parseString str =
   case parse contracts "" str of
     Left e  -> error $ show e
     Right r -> r
```

Figure A.1: Parsec specification of a parser for different types of contracts

Appendix B

Parser for Simplify's output

```
simplifyDef :: LanguageDef st
simplifyDef = emptyDef { P.reservedNames = ["Valid", "Invalid"] }

lexer :: P.TokenParser ()
lexer = P.makeTokenParser simplifyDef

reserved :: String -> Parser ()
reserved = P.reserved Prover.lexer

int :: Parser Integer
int = P.integer Prover.lexer

whiteSpace :: Parser ()
whiteSpace = P.whiteSpace Prover.lexer

validOrInvalid :: Parser Bool
validOrInvalid = do
  optional (PC.char '>')
  Prover.whiteSpace
  Prover.int
  PC.char ':'
  Prover.whiteSpace
  choice [ do { reserved "Valid"; return True }, do { reserved
     "Invalid"; return False } ]

findResponse :: Show a => Prover -> Parser a -> IO a
findResponse prover parser = do
  line <- hGetLine (outH prover)
```

```
case parse parser "" line of
  Right b -> return b
  Left _ -> findResponse prover parser
```

Figure B.1: Parsec specification of a parser for Simplify's responses