Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Wojciech Baranowski
Nr albumu: 277548

Automated verification tools for Haskell programs

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dra Marcina Benke
Instytut Informatyki

Maj 2014

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora (autorów) pracy

Abstract

Presented in this thesis are two recently published verification tools for Haskell programs: Haskell Contracts Checker and LiquidHaskell. They are compared with each other in regard to the verification methods they employ, their expressiveness and performance. In addition, a serious design flaw leading to erroneous behaviour is identified in each of the systems. Finally, several possible improvements and enhancements are suggested.

Słowa kluczowe

functional programming, software verification, Haskell

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

D. Software
D.2. Software engineering
D.2.4. Software/Program Verification

Tytuł pracy w języku polskim

Narzędzia do automatycznej weryfikacji programów w Haskellu

Contents

Introduction

An important part of software development and maintenance is ensuring that the software works correctly. That is, its behaviour matches expectations based on documentation and specification. A popular way of enforcing correctness is testing. This approach, however, is flawed: Almost always exhaustive testing is impossible due to infinite or enormous domain. While non-exhaustive testing in practice may greatly increase confidence in program's correctness, it cannot by itself prove it. Trust level thus obtained may not be sufficient in some applications, such as software controllers for medical or military equipment.

In the past years, formal software verification has been getting more and more attention. It appears, however, that those efforts have been mostly focused on imperative programming languages, most notably C. Among the popular functional languages, Haskell in particular suffers from lack of formal verification platforms. One of the possible reasons is low popularity. According to the TIOBE index[1], Haskell is consistently much less popular not only than the top imperative and objective languages, but also other functional languages, such as Lisp or ML. Another reason might be developers' trust in an extremely powerful type system, statically enforced by the compiler. Thanks to it, programming errors that in other languages would lead to dangerous bugs or vulnerabilities, in case of Haskell often prevent the program from compiling. However, this trust, while not baseless, cannot be fully justified, as there is plenty of programming errors that Haskell type system does not intercept, such as insufficient neutralization of user input used in SQL and shell commands, or integer overflow. All these issues, according to MITRE Corporation, are among 25 most dangerous programming errors[2]. Yet another reason might be laziness of Haskell, which creates some unique problems and renders verification strategies for strict languages inapplicable.

Regardless of its cause, this neglect might be starting to get some attention and focus with two tools for automated verification of Haskell programs having been recently published. This thesis presents analysis of them both. The thesis is organized as follows:

Chapter 1 describes the recently published tools. The main focus is their functionality and syntax, illustrated with numerous examples. Other projects related to verification of Haskell programs are also briefly mentioned, namely: an outdated utility for finding runtime errors, a model-checking tool using Haskell as a front-end language and two testing platforms.

The two verification tools are then compared with each other in chapter 2, in regard to their internal workings, their expressiveness and their performance. Presented are also efforts to verify some simple real-world properties in both the systems. Finally, their usability in both software development and didactics is assessed based on that comparison.

Both tools are hindered by serious design flaws, which lead to erroneous behaviour. Those are explained in chapter 3, together with some minor bugs.

Finally, chapter 4 suggests possible improvements to both the systems. Their expected

[1] `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`
[2] `http://cwe.mitre.org/top25`

benefits include increasing expressiveness, addressing aforementioned design flaws and improving performance of the tools.

Chapter 1

Existing tools for verification of Haskell programs

This chapter presents several tools capable of verifying or testing correctness of Haskell programs. Two verification tools have been published recently, those are discussed in the first two sections and will remain focus of this thesis. The remaining sections briefly present other tools that are either outdated or are related to verification of Haskell programs, but are not actual verification tools.

1.1. Haskell Contracts Checker

The hcc (Haskell Contracts Checker, formerly called halo; [16]) tool allows to express contracts for Haskell programs and automatically verify that the program indeed complies with its contract. It can handle higher-order functions and lazy evaluation, the two features of Haskell that distinguish it from other popular functional languages but also make it hard to analyze.

1.1.1. Definitions

Definition 1.1.1. *Crash-freedom* (*CF*) of an expression `e` means that `e` does not crash. That is, it either evaluates successfully or diverges.

Definition 1.1.2. For a given type `u`, a *contract* of type `Contract u` is a property of expressions of type `u`. The contract itself is not bound to a specific expression.

Examples:

- A contract of any type might be: *The expression is crash-free.*
- A contract of type `Bool` might be: *The expression value is* `True`.
- A contract of type `a -> b` for some `a` and `b` might be: *As long as the argument is crash-free, the entire expression is also crash-free.*

Definition 1.1.3. A *statement* is a property of a Haskell program to be verified.

The most common kind of a statement is: *The expression* `e` *complies with contract* `c`. Other kinds of statements are introduced below.

1.1.2. Syntax and semantics

Below follows a simplified syntax description for contracts and statements. The description is detailed enough to understand all the code samples presented in this thesis and to write one's own specifications to be verified by the hcc tool. Yet it omits many details that might unnecessarily confuse the reader. All of those, including formal grammar and subtle type system constraints, might be easily deduced from the `https://github.com/danr/contracts/blob/master/testsuite/Contracts.hs` file by an experienced Haskell programmer.

The contracts and statements are expressed in Haskell itself. This design decision was made to save users from the necessity of learning another language. As a result, all the contracts and statements are also Haskell expressions. However, the following description makes a distinction between them for the sake of readability: Whenever an expression is known to be a statement or a contract, it's referred to specifically as such. The word *expression* is used to denote expressions that might or might not be contracts or statements (and in almost all applications are not).

Contracts

A contract might be constructed in one of the following ways:

- `CF`

  the `CF` can be thought of as a constant of type `forall a . Contract a` that represents crash-freedom of expressions.

- `Pred e`

  where the type of `e` is `a -> Bool` for some type `a`. The resulting contract's type is `Contract a`.

  The useful way to think about this construct is as a way to turn an ordinary Haskell predicate into a contract.

- `c1 :&: c2`

  where both `c1` and `c2` are contracts of the same type, i.e. they are both of type `Contract a` for some type `a`. The resulting contract's type is also `Contract a`.

  This construct represents conjunction of two contracts.

- `c :-> e`

  where the `c` contract is of type `Contract a` and the `e` expression is of type `a -> Contract b` for some types `a` and `b`. The resulting contract is of type `Contract (a -> b)`.

  As can be deduced from the resulting contract's type, this construct is used to create contracts for functions. An expression `f` of type `a -> b` complies with the resulting contract iff for any expression `x` of type `a` that complies with the contract `c`, the expression `e x` either diverges or evaluates to a contract that the expression `f x` complies with.

  For arguments that do not comply with `c`, the resulting contract does not provide any guarantees.

Example 1.1.1 (Dependent arrow contract)**.** Consider the following contract:

```
CF :-> \x -> CF :&: Pred (>= x)
```

which can be used with any function of type:

```
a -> b
```

with `a` and `b` being such types that the operator `(>=) :: b -> a -> Bool` is defined.
The contract expresses that if the argument is crash-free, then the result is also crash free and is greater or equal to the argument.

- `c1 --> c2`

  where `c1` and `c2` are contracts respectively of types `Contract a` and `Contract b` for some types `a` and `b`. The resulting contract's type is `Contract (a -> b)`.

  This construct is a special case of the function contract, in which the resulting contract does not depend on the argument value other than its compliance with the `c1` contract.

  An expression `f` of type `a -> b` complies with the resulting contract iff for any expression `x` of type `a` that complies with the contract `c`, the expression `f x` complies with `c2`.

Example 1.1.2 (Non-dependent arrow contract)**.** For any unary function the property: *as long as the argument is crash-free, the result is also crash-free* may be expressed with the following contract:

```
CF :-> \_ -> CF
```

However, since the contract for the result does not depend on the argument, the same property may be expressed in a simplified way:

```
CF --> CF
```

Statements

A statement might be constructed in one of the following ways:

- `e ::: c`

  where `e` is an expression of type `a` and `c` is a contract of type `Contract a` for some type `a`.

  The meaning is: *The expression `e` complies with the contract `c`.*

- `s1 :=> s2`

Where `s1` and `s2` are statements.

The `:=>` operator denotes implication: *If the `s1` statement holds, then `s2` also holds.*

- `All e`

  Where the expression `e` is of type `a -> Statement` for some type `a`.

  The `All` keyword denotes universal quantification: *For any expression `x` of type `a`, the expression `e x` either diverges or evaluates to a statement that holds.*

Beside the constructs listed above there is also the `Using` keyword:

```
s1 `Using` s2
```

It does not really create any new statement but indicates a dependency between the `s1` and `s2` statements. It is a way to hint the hcc tool to first try to verify the `s2` statement and only then the `s1` statement.

Examples

All the code samples below will assume the following data definitions:

```
data Nat = Z | S Nat
```

Example 1.1.3.

```
two :: Nat
two = S (S Z)

two_cf = two ::: CF
```

The statement in the last line simply asserts that the `two` expression does not crash.

Example 1.1.4.

```
incr :: Nat -> Nat
incr i = S i
incr_cf = incr ::: CF --> CF

decr :: Nat -> Nat
decr (S i) = i
decr_cf_broken = decr ::: CF --> CF
```

The `incr_cf` statement asserts that as long as the argument is crash-free, the application of `incr` function is also crash-free. However, a similar statement about the `decr` function is not true: It might crash if provided with the (crash-free) `Z` constant as an argument. hcc indeed proves that the `decr_cf_broken` statement is false.

Example 1.1.5. The `decr` function from the previous example is CF as long as the argument is CF and is not equal to `Z`. This can be expressed with the help of a predicate checking that a value is not `Z`. The same predicate can be used to strengthen statement about the `incr` function:

```
positive :: Nat -> Bool
positive Z = False
positive (S _) = True

decr_cf = decr ::: (CF :&: (Pred positive)) --> CF
incr_cf_and_pos = incr ::: CF --> (CF :&: (Pred positive))
```

Example 1.1.6. Other statement constructors can be used to express more complex properties, as in the following sample found in HCC testsuite[1]:

```
reflexive :: (a -> a -> Bool)
          -> Statement (All a (Assuming a Bool))
reflexive (~~) =
    All (\x -> x ::: CF :=> x ~~ x ::: CF :&: Pred id)

(<=) :: Nat -> Nat -> Bool
Z     <= _     = True
_     <= Z     = False
(S x) <= (S y) = x <= y

le_refl = reflexive (<=)
```

The `reflexive` function when applied to any binary relation produces a statement asserting that the relation is reflexive. Thus the `le_refl` statement asserts that the `(<=)` relation defined in the sample is reflexive.

1.2. LiquidHaskell

The *LiquidHaskell* project (formerly called Hsolve, [15]) aims to strengthen Haskell type system. The project stems from a similar system for OCaml, called *LiquidTypes* (short for *Locally Qualified Data Types*; [10]).

It allows to annotate type signature declarations with logical predicates. It can also infer such annotated types to minimize annotation burden put on the user. The predicate language is fixed in such a way that the verification is decidable. As a result, however, the

[1] `https://github.com/danr/contracts/blob/master/testsuite/Properties.hs` and `https://github.com/danr/contracts/blob/master/testsuite/Nat.hs`

expressiveness of the system is severely limited.

Since LiquidHaskell is under active development and newer revisions are very unstable, discussed in this thesis is version 0.1 of the software, which can be obtained from `https://github.com/ucsd-progsys/liquidhaskell/tree/liquidHaskell-0.1`.

1.2.1. Definitions

Definition 1.2.1. *Refined type* is a Haskell type combined with a logical predicate. The predicate is called *refinement*. A bare Haskell type can be thought of as a refined type whose refinement is trivial: A predicate that is true for any value.

Definition 1.2.2. When talking about a refined type, the underlying Haskell type is called *sort*.

1.2.2. Syntax and examples

Since the aforementioned annotations are not expressed in Haskell, LiquidHaskell defines its own syntax. It is introduced below by means of examples with comments and explanations rather than more formal methods. This approach should be more efficient for several reasons:

1. the syntax is quite complicated, with several inconsistencies;
2. it mirrors Haskell syntax and thus reading complete examples should be easy for a reader familiar with the language;
3. formal description of LiquidHaskell syntax has not been published.

Basic refinements

All LiquidHaskell annotations within a Haskell source file are enclosed between `{-@` and `@-}` sequences. That way they are treated as comments by the compiler and do not confuse it.

A refined type declaration is a copy of type signature with refinements inserted into it. Any type or type variable `a` appearing in the original type signature might be refined using the following syntax:

```
{ v:a | f }
```

where `f` is a formula, possibly containing `v` as a free variable. This construct mirrors notation commonly used in mathematics:

$\{v \in \mathbb{N} \mid v \geq 42\}$

In addition, when a function is being specified, any argument might be named, by prefixing its refined type with a name of user's choosing and a colon.

Definition 1.2.3. *Dependent refinement* is a refinement of a functional type that depends on the function's arguments.

Example 1.2.1 (Basic refinement)**.**

```
{-@ id :: x:a -> {v:a | v = x} @-}
id :: a -> a
id x = x
```

The first line contains declaration of the refined type for the `id` function. The first part (`x:a`) declares the sort of the argument to be `a`. It also names the argument as `x`.

The second part (`{v:a | v = x}`) defines the refined type of the result. It states that the result is equal to the argument. Since the name given to the argument appears in the formula as a free variable, it is a dependent refinement.

It's worth to note that the sort of refined type matches the type signature in the program.

Example 1.2.2 (Argument refinement)**.**

```
{-@ divide :: Int -> {v:Int | not(v=0)} -> Int @-}
divide :: Int -> Int -> Int
a `divide` b = a `div` b
```

Produced with the refined type above, LiquidHaskell enforces that any call to `divide` function has a correct refined type. That is, that the second argument passed to `divide` is never `0`.

This example comes from the LiquidHaskell blog[2].

Abstract refinements

The concept of abstract refinements is best explained with an example:

[2] `http://goto.ucsd.edu/~rjhala/liquid/haskell/blog/blog/2013/01/01/refinement-types-101.lhs/`

Example 1.2.3 (Abstract refinements)**.**

```
{-@ max :: forall a <p :: a -> Prop> .
           (Ord a) => a<p> -> a<p> -> a<p> @-}
max :: (Ord a) => a -> a -> a
max x y = if x < y then y
                   else x
```

The liquid type above states that for any type `a` belonging to the `Ord` typeclass and for any possible refinement `p` of that type, if the refinement is true for both the arguments passed to the `max` function then it will be also true for the result.

Just as the type variable `a` will be instantiated with concrete types when the function max is called, so the abstract refinement `p` will be instantiated with concrete refinements by LiquidHaskell.

This example comes from [15].

As can be deduced from the example above, it is possible to use universal quantification over all possible predicates of a given sort when defining a refined type. The syntax extends the one introducing universally quantified types in Haskell. In LiquidHaskell, the `forall` keyword may be followed not only by type variable names but also by predicate names and their sorts. The return type of every predicate must be `Prop`, which is LiquidHaskell's analogue of Haskell's `Bool`. Such predicates are called *abstract refinements*. A comma-separated list of abstract refinements must be enclosed in angle brackets.

To state that a Haskell type `a` is refined by an abstract refinement `p`, the notation:

```
a<p>
```

is used. It can be thought of as equivalent of:

```
{ v:a | p v }
```

However, LiquidHaskell parser does not allow those two notations to be used interchangeably. The former is reserved exclusively for abstract refinements and the latter for concrete refinements.

Existential quantification

LiquidHaskell allows also for existential quantification over values of a given type in refined type declarations. An example[3] will best demonstrate its usage:

[3]From: `http://goto.ucsd.edu/~nvazou/liquidtutorial/Composition.lhs.slides.html`, with slight modifications

Example 1.2.4 (Existential quantification)**.**

```
{-@ compose :: forall a b c < p :: b -> c -> Prop
                            , q :: a -> b -> Prop > .
    f:(argb:b -> c<p argb>) ->
    g:(arga:a -> b<q arga>) ->
    x:a ->
    exists[y:b<q x>] . c<p y> @-}
compose :: (b -> c) -> (a -> b) -> a -> c
compose f g x = f $ g x
```

In the code above, the `compose`'s refinement is parametrized with two abstract refinements. They describe possible properties of the first and second arguments respectively, which are functions to be composed. `x` denotes the argument to which both functions are to be applied in composition. The line:

```
    exists[y:b<q x>] . c<p y>
```

states that there exists `y` such that:

- it has the same refined type as application `g x` and
- the result of the `compose` function has the same refined type as `f y`.

Measures

Measures are LiquidHaskell analogues of Haskell's functions. They allow for expressing basic properties of complex data structures and may be used within predicates (since the predicates may not use Haskell functions) or to reason about termination. They are defined with the use of `measure` keyword. Other than that, the syntax closely resembles that used for defining functions in Haskell.

Example 1.2.5 (Measure)**.**

```
{-@ measure len :: [a] -> Int
            len ([]) = 0
            len (x:xs) = 1 + len(xs) @-}

{-@ append :: l:[a]
           -> m:[a]
           -> {v:[a]|len(v)=len(l)+len(m)} @-}
append :: [a] -> [a] -> [a]
append [] zs = zs
append (y:ys) zs = y : append ys zs
```

The above sample from [15] defines `len` measure which represents length of any list. The measure is then used to verify that `append` produces list whose length is the sum of lengths of the arguments.

It is important to note that a measure may accept only one argument which must be of an algebraic data type and there must be exactly one definition for every data constructor of that type.

Predicates

The keyword `predicate` can be used to define macros to be inlined in refinement formulae.

Example 1.2.6 (Predicate)**.** Using code from the previous example, we can assert that the result of `append` is a non-empty list if the second argument is also non-empty:

```
{-@ predicate NonNull X = len(X) > 0 @-}

{-@ append :: [a]
           -> {v: [a] | (NonNull v)}
           -> {v: [a] | (NonNull v)} @-}
```

Invariants

The keyword `invariant`, as the name suggests, can be used to assert that certain property holds for every instance of a given data structure.

Example 1.2.7 (Invariant)**.** Length of a list is always a non-negative value:

```
{-@ invariant {v:[a] | (len v) >= 0} @-}
```

Data Abstract Refinements

While data abstract refinements have similar syntax to abstract refinements in ordinary type declaration, the purpose they serve is quite different. When defining a new data type, one can use data abstract refinements to specify semantics of refining that particular data type, i.e. what kind of concrete refinements might be defined for it.

Consider the following definition of list data type:

```
data L a
    = N
    | C a (L a)
```

While some characteristics can be expressed using measures, one may need to assert other, more demanding properties. For example, a property that all the members of the list are the same, or unique, or that the list is sorted. Those cannot be expressed with the use of LiquidHaskell predicate language, since its designers cannot anticipate all possible data types declared by the users. Data abstract refinements allow users themselves to overcome this limitation.

Example 1.2.8 (Data Abstract Refinement — List)**.**

```
data L a
    = N
    | C a (L a)

{-@ data L a <p :: a -> a -> Prop>
        = N
        | C (x  :: a)
            (xs :: L (a<p x>) <p>)
        @-}

{-@ type IncrL = L<{\a b -> b > a}> Int @-}
{-@ type DecrL = L<{\a b -> b < a}> Int @-}
```

The example above inspired by [15] contains declaration of a custom list type. The declaration is copied and annotated. The data type is parameterized with one abstract refinement. Any concrete predicate to replace that abstract refinement should accept two arguments, members of the list.

If the data abstract refinement is replaced with a concrete predicate, the predicate is guaranteed to be true for any two distinct members of the list. The first argument passed to the predicate is guaranteed to occur in the list before the second one. To understand why it is so, consider the last line of refined declaration:

```
(xs :: L (a<p x>) <p>)
```

It refers to the tail of the list. The `(a<p x>)` part states that the predicate has to be true when applied first to `x` and then to any element of the tail. But `x` is head of the current list.

The `<p>` part states that the predicate has to be true when applied in the same manner to the tail of the list, i.e. when applied first to the head of the tail and then to any element of the tail of the tail. And so on until the empty list is reached.

The last two lines instantiate this refined declaration with concrete refinements. They define type synonyms that denote lists of `Int`s in increasing and decreasing order respectively.

There are two points worth noting:

- In reality, custom list type declaration is not required as LiquidHaskell comes with annotated versions of many built-in functions and data constructors, including the standard list.
- Since concrete refinements have to be enclosed in angle brackets, they can be easily distinguished from the other arguments to the type constructor. For this reason,

LiquidHaskell parser allows them to appear anywhere within the type constructor application.

Take, for example, the fragment from the above sample:

```
(xs :: L (a<p x>) <p>)
```

It could be also replaced by:

```
(xs :: L <p> (a<p x>))
```

In fact, the latter notation (i.e., placing refinement immediately after type constructor) is prevalent in LiquidHaskell testsuite and for this reason is also frequently used in this thesis.

Data abstract refinements can be instantiated with several different concrete refinements. That way, single data declaration can be reused in many different refined type declarations that may or may not overlap. Similarly to how refining a type was done, refining a data declaration requires one to copy original declaration and then annotate it.

Example 1.2.9 (Data Abstract Refinement — Map)**.**

```
data Map k a = Tip
             | Bin Size k a (Map k a) (Map k a)
{-@
data Map k a < l :: root:k -> k -> Prop
             , r :: root:k -> k -> Prop >
    = Tip
    | Bin (sz :: Size)
          (key :: k)
          (value :: a)
          (left :: Map <l, r> (k <l key>) a)
          (right :: Map <l, r> (k <r key>) a)
@-}
{-@ type OMap k a = Map < {\root v -> v < root }
                        , {\root v -> v > root } > k a @-}
```

The last line declares `OMap` to be a map preserving the BST condition, i.e. any keys in the left and right subtree are respectively strictly smaller and strictly greater than the key in the root. To do so, it instantiates refined `Map` declaration with two predicates, as requested in the refined definition.

The `(k <l key>)` and `(k <r key>)` parts of the declaration indicate that the root's key is passed as the first argument to both the predicates. The `Map <l, r>` parts indicate that those predicates are applied in the same manner to both the subtrees.

The code sample comes from LiquidHaskell testsuite[4].

[4] https://github.com/ucsd-progsys/liquidhaskell/blob/liquidHaskell-0.1/tests/pos/Map.hs

As can be seen in the two examples above, the syntax for instantiating a data abstract refinement is similar to that of lambda expressions in Haskell. The `\` symbol is followed by arbitrary names of arguments, then by an arrow (`->`) and then by a formula, possibly using the arguments as free variables. Every data abstract refinement must be enclosed in braces.

Sets

LiquidHaskell is capable of reasoning about sets of values. To this end it introduces the `Set` type and several measures operating on it[5]:

```
-- | union
measure Set_cup  :: (Set a) -> (Set a) -> (Set a)

-- | intersection
measure Set_cap  :: (Set a) -> (Set a) -> (Set a)

-- | difference
measure Set_dif  :: (Set a) -> (Set a) -> (Set a)

-- | singleton
measure Set_sng  :: a -> (Set a)

-- | emptiness test
measure Set_emp  :: (Set a) -> Prop

-- | membership test
measure Set_mem  :: a -> (Set a) -> Prop

-- | inclusion test
measure Set_sub  :: (Set a) -> (Set a) -> Prop
```

Names of the measures and comments above them should be self-explanatory. Is is important to note that even though LiquidHaskell's `Set` in many ways resembles the data type defined in `Data.Set` module, they are two separate constructs.

[5]Source: `https://github.com/ucsd-progsys/liquidhaskell/blob/liquidHaskell-0.1/include/Data/Set.spec`

Example 1.2.10 (Reasoning about sets)**.**

```
{-@ measure lMembers :: [a] -> (Set a)
            lMembers([]) = {v | Set_emp(v) }
            lMembers(x:xs) = {v |
                v = (Set_cup (Set_sng x) (lMembers xs) ) }
            @-}
{-@ append :: l:[a]
              -> k:[a]
              -> {v: [a] | (lMembers v) =
                                 (Set_cup (lMembers l)
                                          (lMembers k) ) } @-}
append :: [a] -> [a] -> [a]
append [] zs = zs
append (y:ys) zs = y : append ys zs
```

The `lMembers` measure when applied to a list returns a set of its elements. It is used to express the following refinement of `append`'s type: The set of resulting list's elements is a sum of sets of elements of the arguments. In other words, the result contains all the elements that either of the arguments contains.

Termination

While verifying type refinements is the main functionality of LiquidHaskell, the tool is also capable of verifying termination of recursive functions. It does so by proving that one of the numerical arguments to the function decreases with every recursive call. The argument might be also of an algebraic data type, as long as there is a measure defined for it that produces an integer.

By default, the first such argument is considered, but the user might customize this behaviour using the `Decrease` keyword.

Example 1.2.11 (Proving termination)**.**

```
{-@ Decrease append 1 @-}
append :: [a] -> [a] -> [a]
append [] zs = zs
append (y:ys) zs = y : append ys zs
```

LiquidHaskell will prove that `append` function terminates because the first argument decreases with every recursive call (or rather, the `len` measure, which is a default measure used for lists, applied to the first argument decreases with every call). Because the first argument is also the first argument that is either numeric or has a numeric measure defined, the `Decrease` declaration is optional: LiquidHaskell would have assumed it anyway.

Had the user declared `Decrease append 2` instead, LiquidHaskell would have produced an error since the second argument remains unchanged in every recursive call.

The measure to be used for a given algebraic data type is declared in the refined data type definition.

Example 1.2.12 (Default measure for an algebraic data type)**.**

```
{-@
  data Map [mlen] k a < l :: root:k -> k -> Prop
                      , r :: root:k -> k -> Prop>
       = Tip
       | Bin (sz    :: Size)
             (key   :: k)
             (value :: a)
             (left  :: Map <l, r> (k <l key>) a)
             (right :: Map <l, r> (k <r key>) a)
  @-}

{-@ measure mlen :: (Map k a) -> Int
    mlen(Tip) = 0
    mlen(Bin s k v l r) = 1 + (mlen l) + (mlen r)
  @-}
```

In the sample above[6] measure `mlen` is declared as the default measure for the `Map` data type. That means that whenever LiquidHaskell tries to prove termination of a recursive function using an argument of that type, it will try to prove that every recursive call is performed with a map for which this measure's value is smaller than for the original argument.

Termination checking may be disabled by running the tool with a `--notermination` option.

Totality

When passed the `--totality` option, the tool also reports errors for every incomplete pattern matching. A function domain might be limited with refined types and so pattern matching need not necessarily be exhaustive.

Example 1.2.13 (Incomplete pattern matching)**.**

```
{-@ incomplete :: {v:Bool| v=True} -> Bool @-}
incomplete :: Bool -> Bool
incomplete True = True

{-@ incomplete2 :: {v:Bool| v=False} -> Bool @-}
incomplete2 :: Bool -> Bool
incomplete2 True = True
```

When ran with `--totality` option, LiquidHaskell reports an error for `incomplete2` function but not for `incomplete`.

1.3. Other tools

1.3.1. Catch

The Catch project ([9]) aims at verifying that a Haskell program does not crash, but not its functional correctness. The tool works by:

1. For every non-exhaustive pattern-matching, implicitly adding missing patterns and defining the corresponding expression to call the `error` function.
2. Identifying all calls to the `error` function.
3. Computing preconditions on every function's arguments that prevent any call to the `error` function from being reached.

The tool has two major shortcomings:

1. The preconditions might be more restrictive than necessary. As a result, Catch might report false errors.
2. The tool is able to reason only about algebraic data types. It thus simplifies integer data types with the following definition:

   ```
   data Int = Neg | Zero | One | Pos
   ```

[6] Taken from LiquidHaskell testsuite: `https://github.com/ucsd-progsys/liquidhaskell/blob/liquidHaskell-0.1/tests/pos/Map.hs`

Despite that, it was tested against real-life projects and used in development cycle of at least one of them. The results presented in [9] look indeed very promising.

The project was, however, discontinued due to termination of a project it depended on. Even though the source code is still available, nowadays it is virtually impossible to build the package. For this reason the project is not further discussed in this thesis.

1.3.2. Zeno

Quite opposite to Catch, Zeno ([13]) is capable of of verifying functional correctness of a program but it requires all the values defined in the program to be total, i.e. not crashing or diverging.

It verifies programs written in a small subset of Haskell, called HC. The specification language (called PHC) is fairly simple. It can be used to express:

- Equality of two HC expressions.
- Implication of the form:

  $e_1 = e_2 \wedge e_3 = e_4 \wedge \ldots \wedge e_i = e_{i+1} \Rightarrow e_{i+2} = e_{i+3}$

It also allows for using universal quantification but only at the topmost level and it does not allow for existential quantification. As a result, it is strictly weaker than FOL.

It appears that Zeno is not meant to be a verification tool for Haskell, but rather a model checking tool for verifying functional programs that happens to be using a subset of Haskell as a front-end. For this reason, Zeno is also not further discussed in this thesis.

1.3.3. Property-based testing tools

Reader familiar with testing tools such as QuickCheck ([3]) or SmallCheck ([11]) might find HCC syntax familiar. Those utilities allow for describing Haskell functions with so called *properties*, which consist of logical predicates. The predicates can be written in Haskell and may consist of arbitrary boolean-valued expressions, as is the case with HCC. What is more properties may use not only predicates, but also quantifiers and implication.

The key difference between those two tools and HCC, is that instead of verifying they *test* properties by either generating random test cases (QuickCheck) or fully searching domain up to a user-provided depth (SmallCheck). In combination with Haskell Program Coverage toolkit ([7]) they might provide great confidence in program correctness. Nevertheless, since they do not formally verify programs, they are not further discussed in this thesis.

Chapter 2

Comparison of Haskell Contracts Checker and LiquidHaskell

This chapter compares and further analyzes Haskell Contracts Checker and LiquidHaskell, the two verification tools introduced in the previous chapter. First verification methods employed by both systems are discussed in section 2.1. Then they are compared against each other in regard to their expressiveness in section 2.2 and their performance in section 2.3. Finally, section 2.4 describes efforts to write and verify some real-world yet simple specifications in both the systems.

2.1. The pipeline

The comparison of both systems begins with presenting their pipelines. That is, their verification procedures, including all the custom intermediate languages and translating tools. This description should provide enough information about the inner workings to understand limitations and bugs of both systems, discussed in the next sections and the next chapter.

2.1.1. Overview of HCC pipeline

Figure 2.1 presents simplified pipeline employed by HCC. The source code is first translated to the Core language, using API provided by a Haskell compiler. It is then further simplified and translated to a language called λ_{HALO}. Each statement exported by the source file is then translated to first order logic, together with all the program definitions relevant to it, and placed in its own file. All such files are then checked by a theorem prover.

2.1.2. Overview of LiquidHaskell pipeline

Figure 2.2 presents LiquidHaskell's pipeline. The source code is translated to the Core language, but the source file is also parsed to obtain all the refinements provided by the user. Both the Core representation and refinements are then used to produce a file in a format internally called *Fixpoint*. The file contains refinements provided by the user and describes dependencies between refinements of expressions present in the original program. A tool called fixpoint is then provided with that file to produce several FOL theories, each corresponding to a different possible set of concrete refinements inferred for every subexpression in the program. All those theories are placed in a single file which is then checked by the Z3 theorem prover [4].

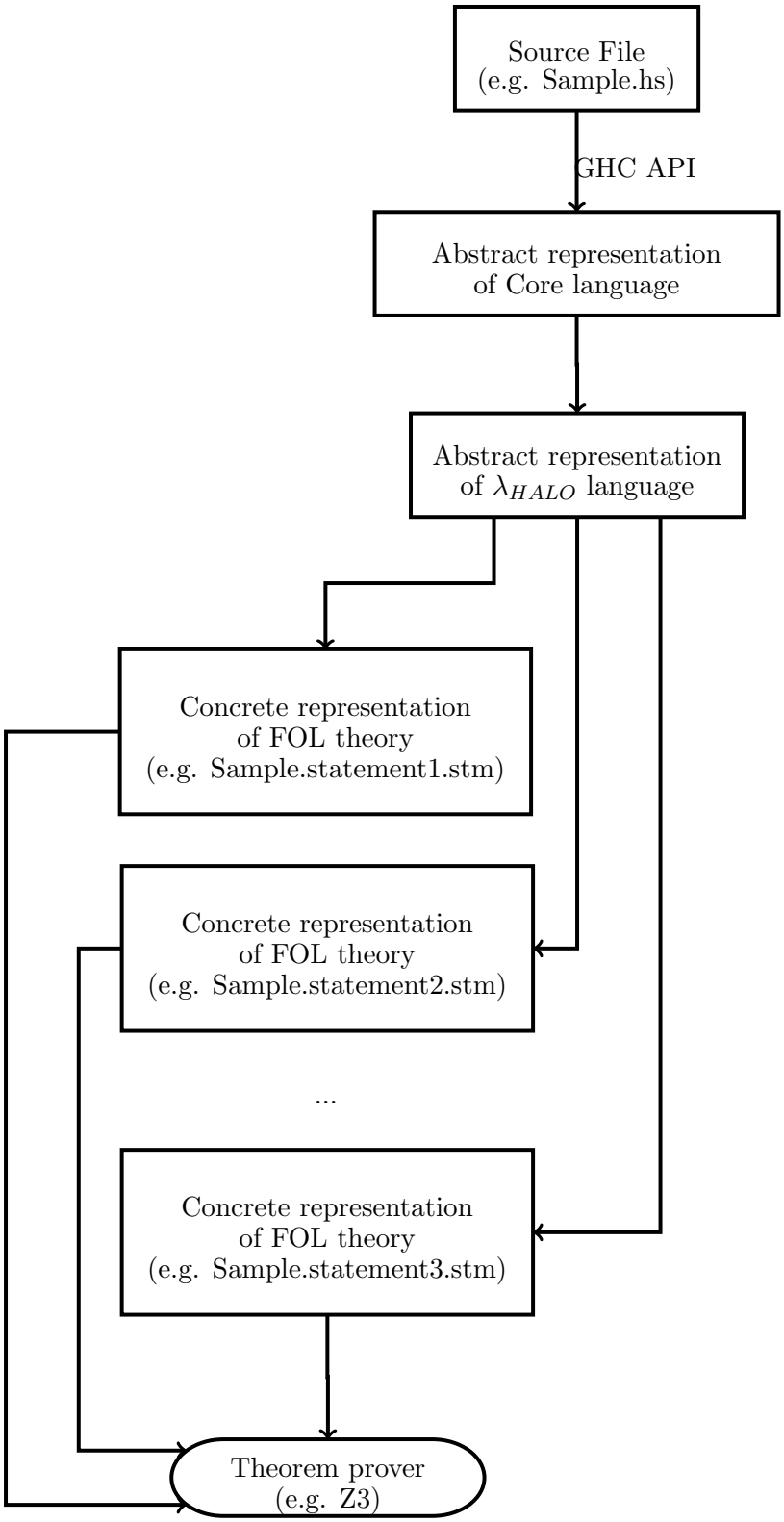

Figure 2.1: HCC pipeline

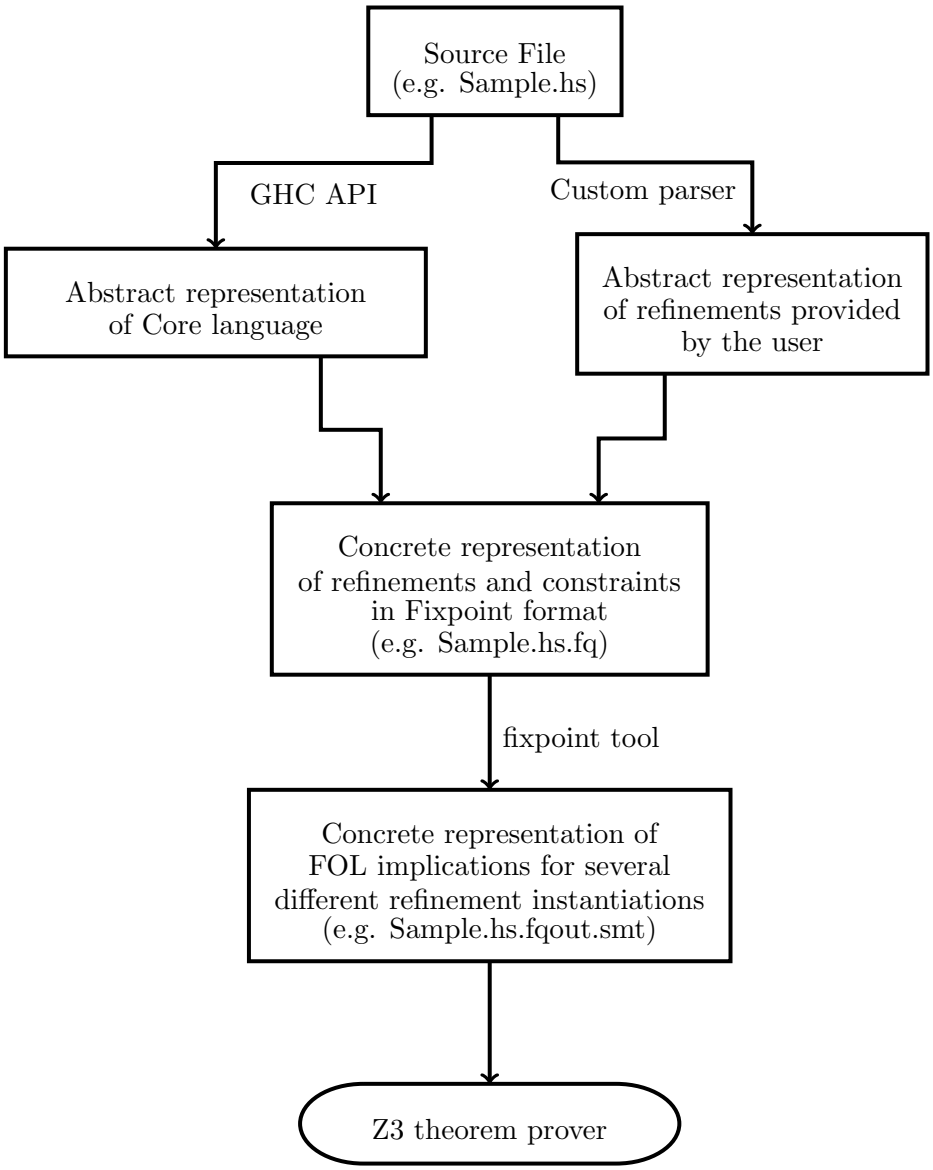

Figure 2.2: LiquidHaskell pipeline

2.1.3. Parsing the source file

HCC uses GHC (Glasgow Haskell Compiler) API to parse the Haskell program and translate it to Core, an intermediate, greatly simplified language used by the compiler. On top of that, HCC further simplifies the code, thus translating the original program to the λ_{HALO} language introduced in [16]. At this stage contracts and statements are not treated differently from other definitions in the program.

LiquidHaskell also uses GHC API to obtain Core translation of the original program. Contrary to HCC, however, it also parses the source file on its own, specifically, all the LiquidHaskell declarations enclosed between `{-@` and `@-}` markers which are translated to an abstract internal representation.

2.1.4. The intermediate languages

Both the systems introduce custom intermediate languages that the original programs get translated to. Those languages, however, are vastly different since they serve different purposes, as will become evident in the following description.

λ_{HALO}, the HCC's intermediate language

The intermediate language used by HCC, λ_{HALO}, has been described in depth in [16]. Presented in this subsection is a brief summary.

λ_{HALO} is a small subset of the Haskell language. The most notable limitations are:

- It supports only algebraic data types.
- It does not support typeclasses.
- It does not support lambda expressions.
- Pattern matching may occur only at the topmost level of a function definition.
- It does not allow for partial application of data constructors.

The latter three limitations do not impose any restrictions on the original program. HCC takes care of them in the process of translation from Core to λ_{HALO} by means of lambda- and case-lifting. The former two, however, must be put also on the input program.

λ_{HALO} behaves like Haskell in other important aspects. That is, it is purely functional and lazily evaluated.

Fixpoint, the LiquidHaskell's intermediate language

The intermediate language used by LiquidHaskell is internally called *Fixpoint*. Unlike λ_{HALO}, it is not a programming language but a format for expressing two kinds of constraints on refined types:

- Well-Formedness constraint is put on a single refined type. It enforces that the refinement is a valid refinement for the sort of refined type.

Example 2.1.1 (Well-formedness constraint)**.** The refined type `{ v:Int | v > 0}` is well-formed. The refined type `{ v:[Int] | v > 0}`, however, is not since lists cannot be compared with integers.

- Subtyping constraint is put on two refined types. It enforces that the first type is a subtype of the second type (hereafter called *supertype*). What it means, is that sorts of both refined types are the same and the refinement of the first one implies the refinement of the second one.

  The notation employed to state that `{ v:sort | refinement1 }` is a subtype of `{ v:sort | refinement2 }` is:

  ```
  { v:sort | refinement1 } <: { v:sort | refinement2 }
  ```

Example 2.1.2 (Subtyping constraint)**.** Assume environment in which:

```
x, y :: { v:Int | v > 0 }
```

Then `{ v:Int | v = x + y }` is a subtype of `{ v:Int | v > 1 }`, because their sorts are the same and the former refinement implies the latter.

To specify those constraints, Fixpoint also has a notion of *environment*. In context of Fixpoint files, an environment is a mapping of variable names to their refined types. Each constraint is set in its own environment and the refinements under the constraint may reference variables from the environment.

It is also possible to use integer arithmetic and uninterpreted functions in Fixpoint files. However, there is no notion of a program or source code.

As to the refinements in refined types, they can be either of the two:

- A concrete refinement, most likely if the user provided the entire refined type in the specification.
- A *liquid variable*, possibly with *pending substitutions*.

  A liquid variable is a placeholder variable to be replaced by a concrete refinement later on.

  Pending substitution to a liquid variable is a pair of variable name and refined type that corresponds to it. Pending substitution is performed once a concrete refinement is assigned to the liquid variable.

  A very detailed explanation may be found in [10].

Generating the Fixpoint file

While translation to λ_{HALO} performed by HCC is quite trivial, the translation performed by LiquidHaskell requires some explanation. First, it is important to understand the purpose of translation. It should be easiest to explain with an example:

Example 2.1.3. Let's assume that LiquidHaskell tries to prove the following specification:

```
{-@ type PosInt = { v:Int | v > 0 } @-}
{-@ max :: Int -> PosInt -> PosInt @-}
max x y = if x > y then x else y
```

In order to prove that the specification is met, LiquidHaskell needs to prove that the refined type of expression defining the `max` function is a subtype of the specified refined type of the result. That is, in the environment:

```
x :: Int
y :: PosInt
```

the expression `if x > y then x else y` has a refined type `rtype1` such that:

```
rtype1 <: PosInt
```

But the expression itself evaluates to value of one of the expressions: `x` and `y`. LiquidHaskell thus needs to prove that (or rather find refined types such that) refined types of those expressions are in certain circumstances subtypes of `rtype1` which in turn is a subtype of `PosInt`.

To the best of author's knowledge constraint generation performed by LiquidHaskell has not been publicly documented anywhere. One may find, however, a detailed description of constraint generation in Liquid Types, a similar tool for OCaml, in section 2.3.2 of [10]. Shared codebase of both projects and analysis of intermediate files produces by LiquidHaskell suggests that the latter uses a similar procedure.

The constraint generation procedure accepts an environment as an argument and for a given expression it produces its refined type and a (possibly empty) set of constraints. It works recursively over an abstract representation of the expression. To provide intuitive understanding of how the procedure works, presented below is explanation for two types of expressions. A reader interested in details may consult [10].

- Application:

  `e1 e2`

  The constraint generation proceeds as follows:

  1. Recursively obtain the type `x:Tx -> T` and set of constraints `C1` for `e1`. (The type is guaranteed to be of the same sort as `Tx -> T`, otherwise the type checking performed by the compiler would have failed.)
  2. Recursively obtain the type `Tx2` and a set of constraints `C2` for `e2`.
  3. The resulting type is T with appropriate substitution for `x`: `T[x := Tx2]`. The resulting set of constraints is union of `C1`, `C2` and the subtyping constraint: `Tx2 <: Tx`.

- If-then-else:

  `if b1 then e1 else e2`

The constraint generation proceeds as follows:

1. Generate a new refined type `T` with the refinement represented by a liquid variable. (The sort of the refined type is obtained from type inference performed by the compiler.)
2. Recursively obtain the set of constraints `Cb` for expression `b1`.
3. Generate a new environment Γ_1 by adding to the current environment the information that `b1` is true. In that environment recursively obtain the type `T1` and set of constraints `C1` for `e1`.
4. Generate a new environment Γ_2 by adding to the current environment the information that `b1` is false. In that environment recursively obtain the type `T2` and set of constraints `C2` for `e2`.
5. The resulting type is `T`. The resulting set of constraints is union of `Cb`, `C1` and `C2` with the following constraints added:
   - `T` is well-formed.
   - In environment Γ_1, `T1` is a subtype of `T`.
   - In environment Γ_2, `T2` is a subtype of `T`.

In general, refinements of almost all expressions are either provided by the user or represented by a liquid variable, to be replaced by a concrete refinement in the inference procedure. If a refinement provided by the user includes an abstract refinement, a concrete refinement to replace it needs to be inferred at every call site. To achieve that, for each call site, a liquid variable is created to represent the concrete instantiation.

2.1.5. Translating λ_{HALO} to FOL

The HCC tool translates abstract representation of the λ_{HALO} language to first order logic, using either the TPTP ([14]) or SMT 2.0 ([2]) formats. Each statement exported by the original Haskell module is translated to FOL, negated and put in a separate file to be processed by a theorem prover. In addition to the negated statement, the file contains also translation of the program, but only definitions that are relevant to that specific statement are included. The resulting theory is then checked by a prover for satisfiability. The original statement holds iff the theory is not satisfiable.

While the details of the translation to FOL of a subset of the statement and contract constructs, as well as proof of soundness, are well explained in [16], presented below is brief summary crucial to understanding advantages and shortcomings of the system.

The *bad* constant

There are two constants used in the signature: *bad* and *unr*. The former represents a crash. Translation of any application of the `error` function produces a term that is equal to *bad*. Also, whenever there is an incomplete pattern matching in the program, the missing patterns are implicitly added during the translation in such a way that the corresponding values are also equal to *bad*. As a result, an expression crashes if and only if its translation is equal to *bad*.

Dropping type information and the *unr* constant

unr stands for *unreachable*. The constant represents two kinds of situation: when an expression diverges or when it is ill-typed. While the former is quite straight-forward, the latter requires some more explanation.

Since the translation is performed after the type-checking took place, the program is guaranteed to be well-typed. That's why the type information can be safely dropped and that's indeed what happens. Now consider the following example[1]:

```
data A = X
data Bool = True | False

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

After the type information is dropped, there is no way to tell that the `not` function can be applied only to the `True` and `False` values, which might lead to errors, for example if there is universal quantification over all possible arguments to `not`. That's why translations of any application of `not` to any other value are explicitly defined to be equal to *unr*.

Functions and function application

A function application is represented by the $app(\cdot, \cdot)$ operation in the signature. The expression `e1 e2` gets translated as $app(t_1, t_2)$ where t_1 and t_2 are translations of `e1` and `e2` respectively.

For any function definition `f x1 x2 ... xN = e` in the program, hcc introduces into the signature an operation $f(\ldots)$ of the same arity and a constant f_{ptr}. The operation is defined in a natural way:

$f(x_1, x_2, \ldots, x_N) = t$

where t is the translation of the expression `e` and x_i is the translation of the name `xi`. The relationship between the f_{ptr} symbol and the operation is defined in following way:

$\forall_{x_1,x_2,\ldots,x_N} app(\ldots(app(app(f_{ptr}, x_1), x_2)\ldots, x_N) = f(x_1, x_2, \ldots, x_N)$

Induction

Whenever HCC encounters a recursive function (or other recursive value) `f` that is provided with a contract `c`, it replaces its identifier with two other. For this brief explanation let them be `f1` and `f2`. `f1` is defined just as `f`, with all calls to `f` replaced by calls to `f2`. HCC then tries to prove that if `f2` meets contract `c`, then also `f1` meets `c`. This approach is generalized and used also for mutually recursive functions.

Statements and contracts

Translation of statements and contracts is mostly obvious and straight-forward. One exception is treatment of the *unr* constant. A compliance with a contract statement (`e ::: c`) is defined to be true if translation of the expression `e` is equal to *unr*. It is also defined to be true if translation of the contract `c` is equal to *unr*, which might be the case with a predicate contract (one defined with an arbitrary Haskell expression) that diverges.

[1] Inspired by a similar example in [16]

2.1.6. Translating Fixpoint to FOL

Part of the LiquidHaskell package is the fixpoint tool which is responsible for solving constraints described in a Fixpoint file.

Constraint solving

The aim of type inference procedure is to replace every liquid variable with a concrete refinement in such a way that all well-formedness and subtyping constraints hold. To generate refinements, fixpoint uses a fixed set of qualifiers.

Definition 2.1.1. A *qualifier* is a logical expression, possibly with some free variables replaced by a placeholder variable `*`. When qualifier is used as part of a refinement this placeholder variable is in turn replaced by a variable from the environment.

fixpoint first initializes all liquid variables with conjunction of all possible instances of all qualifiers, that is, qualifiers with `*` replaced by variables present in the environment. Of those instances, the ones that are not well-formed get discarded.

In the next step, the tool iteratively weakens the refinements just created. It picks any unsatisfied subtyping constraint such that the refinement of the supertype is represented by a liquid variable. The refinement assigned to that liquid variable is then weakened until the constraint is satisfied, i.e. until refinement of the subtype implies refinement of the supertype. This procedure is repeated until all the constraints are satisfied.

If at any point an unsatisfied subtyping constraint is found with a pre-set refinement of the supertype (as opposed to it being represented by a liquid variable), it means that the set of constraints cannot be satisfied and fixpoint reports failure.

Qualifiers to be used when constructing refinements might come from three sources:

- Several of them are built in or, more precisely, included in a small library of *specification* files that comes with LiquidHaskell and beside qualifiers, contains also refined types of several built-in Haskell functions.
- Whenever a user provides a concrete refinement, the formula is turned into a qualifier to be tried with other expressions, by replacing all free variables with placeholder variables.
- A user might explicitly provide some qualifiers using the `qualif` keyword.

The actual translation

Once all liquid variables have been instantiated with some well-formed concrete refinements, fixpoint needs to verify that several implications hold. Specifically, for every subtyping constraint of the form:

```
{ v:sort | refinement1 } <: { v:sort | refinement2 }
```

fixpoint has to verify that `refinement1` implies `refinement2` (in the environment in which the constraint is set). To this end, it translates all such implications with corresponding environments to the SMT 2.0 format ([2]). The resulting file is in turn verified by the Z3 ([4]) SMT prover.

The prover is executed only once, on a SMT file that contains all possible solutions to the constraints set (mappings of liquid variables to concrete refinements). The `push` and `pop` SMT instructions are used extensively to separate possible solutions from each other and denote them all as targets to be checked.

2.1.7. The resulting SMT files

While both HCC and LiquidHaskell are capable of using SMT 2.0 as the back-end format, the files they produce are very different.

The theories and SMT files produced by HCC closely resemble structure of the input program. With a bit of experience, one can rewrite the original program based solely on the SMT file produced by HCC. LiquidHaskell, on the other hand, produces files that provide no hints as to the structure of the original program. This is partially due to the fact that information about program structure is dropped during translation to Fixpoint, and partially because the need to describe all possible constraint solutions makes the output files significantly bigger than the input program.

There are also some differences between type theories employed in resulting files. The FOL theories produced by HCC are unsorted. Even the `True` and `False` values are treated as ordinary constants (or rather ordinary applications of data constructors to empty argument lists). LiquidHaskell, however, takes full advantage of the built-in sorts `Int` and `Bool`, and defines its own sort `Set`.

In addition, LiquidHaskell makes extensive use of uninterpreted function symbols, which represent abstract refinements and data constructors. HCC, on the other hand, provides concrete definitions of virtually all function symbols.

2.2. Expressiveness

Having understood the basics of methods employed by HCC and LiquidHaskell, it is possible to analyze expressiveness of both systems and differences between them in this regard. The following section first discusses which of HCC constructs may be expressed and verified using LiquidHaskell. Then a similar comparison is performed, expressing LiquidHaskell constructs with HCC.

2.2.1. Expressing HCC primitives in LiquidHaskell

HCC syntax can be conveniently thought of as consisting of two layers, the topmost layer being statements, which in turn consist of other statements and the lower layer constructs: contracts. The following analysis of expressiveness is therefore split in two parts, following that structure.

Statements

The *compliance with a contract* statement (`:::`) can be trivially expressed in LiquidHaskell by declaring a refined type for a value. That is, a HCC statement:

```
e ::: contract
```

is roughly equivalent to LiquidHaskell declaration:

```
{-@ name :: rtype @-}
name = e
```

where `name` is an arbitrary identifier, `rtype` is a translation of `contract` and `e` is an arbitrary Haskell expression.

However, while it is possible to specify several contracts for a value in HCC, LiquidHaskell allows every value to have only one refined type. It is, of course, possible to declare more refined types by defining values that are in fact aliases:

```
{-@ name :: rtype1 @-}
name = ... -- definition

{-@ name' :: rtype2 @-}
name' = name
```

This approach, however, poses two problems:

- It obscures the code, which increases the risk of programmers' errors and the maintenance cost.
- When proving a refinement of an expression that uses `name`, only one of the `name`'s refinements will be used. As a result, a programmer needs to understand the refined types inference procedure and deliberately choose one of the `name` versions every time it is used in the code.

This issue is somewhat, but not fully, addressed by LiquidHaskell's abstract refinements.

In LiquidHaskell, this kind of statement is a topmost construct. That is, a refined type declaration for a specific value cannot be used in another construct introduced by LiquidHaskell. For this reason neither implication (`:=>`) statement nor universal quantification statement (`All`) have equivalents in LiquidHaskell.

Contracts

- There is no equivalent for the crash-freedom contract (`CF`) in LiquidHaskell. However, similar functionality can be achieved by:
  - Passing `--totality` option to LiquidHaskell to find all non-exhaustive pattern matches, and
  - refining the type of `error` function in the following way:

    ```
    {-@ assume Prelude.error :: {v:[Char] | false} -> b @-}
    ```

    That is, `error` may not be applied to any argument (because for no argument will the `false` predicate be true) and so can never be called.
- There is also no equivalent for the `Pred e` contract, that is: a contract that uses an arbitrary boolean-valued Haskell expression. Such a construct cannot be found or introduced to LiquidHaskell because its predicate language, by design, is not Turing-complete.
- The arrow contract (`:->`) can be trivially translated to LiquidHaskell by naming arguments in refined type declaration of a function. That is, a HCC contract:

  ```
  contract1 :-> (\x -> contract2)
  ```

  is roughly equivalent to refined type:

  ```
  x:rtype1 -> rtype2
  ```

  where `rtype1` and `rtype2` are translations of `contract1` and `contract2` respectively, and both `contract2` and `rtype2` might use `x` as a free variable.

  As a result, the simplified arrow contract `-->` (which is a special case of `:->`) can be also easily translated to LiquidHaskell.
- The conjunction contract (`:&:`) can be easily translated to LiquidHaskell which allows for using conjunction operator (`&&`) in its predicate language.

2.2.2. Expressing LiquidHaskell primitives in HCC

Unlike HCC, LiquidHaskell syntax does not appear to have clear structure. For this reason, the following description simply enumerates most of LiquidHaskell primitives in an arbitrary order.

Refined type declarations

Refined type declaration can be trivially translated as a statement of compliance with a contract (`:::`).

Concrete refinements

Concrete refinements can be easily translated as boolean-valued Haskell expressions to formulate a contract using the `Pred` construct.

Function types

Function types are equivalent to arrow contracts (`:->` and `-->`). There is, however, certain check performed by LiquidHaskell that HCC does not provide. Whenever a function with a refined type is being called anywhere in the program, LiquidHaskell verifies that the arguments have appropriate refined types, otherwise an error is produced. This behaviour allows to limit functions domains and statically enforce those limits.

It is quite different from HCC semantics. As mentioned in section 1.1.2, HCC does not provide any guarantees for a function when the arguments passed to it do not comply with their contracts. There is no way to limit functions domains using HCC.

Measures and predicates

Measures and predicates can be translated simply as Haskell functions.

Invariants

Invariants can be expressed using the universal quantification. The invariant:

```
{-@ invariant { v:sort | formula } @-}
```

can be translated as:

```
inv_statement = All (\v -> v ::: contract)
```

where `contract` is translation of `formula` and both `contract` and `formula` might use `v` as a free variable.

Abstract refinements

Abstract refinements can also be expressed using universal quantification.

Example 2.2.1 (Translating abstract refinements to HCC)**.** The refined type declaration in:

```
data Nat = Z | S Nat

(>) :: Nat -> Nat -> Bool
Z > _ = False
_ > Z = True
(S x) > (S y) = x > y

{-@ max :: forall < p :: Nat -> Prop > .
           Nat<p> -> Nat<p> -> Nat<p> @-}
max :: Nat -> Nat -> Nat
max x y = if x > y then x else y
```

can be translated to HCC as the following statement:

```
max_quantif = All (\p ->
    max ::: CF :&: (Pred p)
        --> CF :&: (Pred p)
        --> CF :&: (Pred p) )
```

It is worth noting that while this statement is an equivalent of the refined type above, HCC is unable to verify it within 1 hour timelimit on a modern PC. What is more, it is quite hard to write quantification such as the one above without the risk of their verification being affected by a HCC bug discussed in section 3.2.1.

Data Abstract Refinements

Data abstract refinements do not increase system expressiveness, but rather are a way to work around limits of LiquidHaskell's syntax. For this reason there is no need to seek their equivalent in HCC. Expressing predicates on any algebraic data type in the latter is trivial, since it uses Haskell for its predicate language.

Existential quantification

HCC does not explicitly introduce existential quantification and so there is no equivalent of the `exists` keyword in the syntax. However, the implication and universal quantification constructs allow for expressing existential quantification.

Do define one, user might first define a statement that is false:

```
false_stmt = True ::: (Pred $ const False)
```

the statement above is false because no non-divergent value may comply with a false contract. Any non-divergent value other than `True` might have been used to define such statement.

Then one might use the implication constructor (`:=>`) to define `not` operator for statements:

```
not_stmt s = s :=> false_stmt
```

Finally, one might negate universal quantification constructor (`All`) and the corresponding statement to obtain existential quantification:

```
exists f = not_stmt $ All not_f
    where
    not_f x = not_stmt $ f x
```

Example 2.2.2 (Existential quantification in HCC)**.**

```
two_exists = exists $ \x -> ((x == two) ::: CF :&: (Pred id))
```

The statement above asserts that there exists such `x` that when compared to `two`, the comparison does not crash and is true. The sample above omits definitions of `two` and the equality comparison function (`==`).

While the construction above is quite straight-forward, it does not behave as might be expected, due to a bug within HCC design itself, as explained in section 3.2.1. With some non-intuitive modifications, however, it can be fixed.

Termination

Unlike LiquidHaskell, HCC syntax does not include explicit constructs to verify program termination and reasoning about termination might seem impossible in HCC. Especially that, as mentioned in section 2.1.5, every divergent value is considered compliant with every contract. In fact, this behaviour can be used to specify divergence since only divergent values comply with the false contract `CF :&: (Pred $ const False)`. Using the `not_stmt` operator from the previous subsection, one can formulate a statement asserting that a given function does not comply with the false contract and thus either crashes or terminates.

Example 2.2.3 (Reasoning about termination)**.**

```
inf :: Nat -> Nat
inf x = inf (S x)

half :: Nat -> Nat
half (S (S x)) = S (half x)
half _ = Z

inf_divergent = inf ::: CF --> (CF :&: (Pred $ const False))
inf_ends_broken = not_stmt inf_divergent
half_divergent_broken =
    half ::: CF --> (CF :&: (Pred $ const False))
half_ends = not_stmt half_divergent_broken
```

In this sample, `inf` always diverges and `half` always terminates. HCC is able to successfully verify that both `inf_divergent` and `half_ends` statements are true, as expected.

2.2.3. Conclusions

All the primitives of LiquidHaskell language can be also expressed in HCC syntax. The reverse is not true: There are some primitives in HCC that cannot be fully expressed in LiquidHaskell. Those include implication and universal quantification. This is because in LiquidHaskell refined type declaration is the topmost construct, whereas in HCC equivalent of such declaration can be used as part of a more complex statement.

Another HCC primitive not expressible in Liquid Haskell `Pred`, the predicate contract. HCC allows for any boolean-valued function to be used as a contract, which obviously introduces undecidability. LiquidHaskell, on the other hand, uses a predicate language that is not Turing-complete, sacrificing expressiveness.

However, a significant difference in semantics must be noted: LiquidHaskell enforces that for any function, refined types of its arguments are respected at all call sites. HCC makes no such checks. For this reason, HCC cannot be considered strictly more expressive than LiquidHaskell. It is also worth noting that while all the LiquidHaskell constructs can be expressed with HCC syntax, using the latter for purposes it was not designed for requires good understanding of HCC semantics, especially when reasoning about termination.

It appears that HCC might be better suited for reasoning about partial correctness and complex properties of Haskell functions in isolation. In contrast, LiquidHaskell is capable of expressing termination and properties of functions in a broader context, although the properties themselves are limited by the LiquidHaskell predicate language.

2.3. Performance

The following section presents enormous differences between HCC and LiquidHaskell in regard to their efficiency, verification time and decidability.

2.3.1. HCC

HCC source code repository[2] contains a collection of samples with over 340 statements for testing purposes, most of which are very simple (e.g., stating only that a function is crash-free or that it retains length of a list passed to it as an argument). Examples of the most complex ones are presented below. All of those statements have been checked with Z3 and CVC4 ([1]) SMT provers for this thesis.

The tool can use either SMT 2.0 or TPTP format as a back-end and so any other prover supporting one of these could be used to verify statements. In fact, the authors have tested Z3 along with several other provers but the results they published in [16] suggest that Z3 shows significantly better performance than any other prover they tried. They have not, however, tested with CVC4.

Both provers were given 60 seconds to check each of theories corresponding to one of the statements[3]. In practice, however, they always either exceeded this timelimit or produced the answer instantly. This behaviour, albeit surprising, seems to be consistent with the results reported in [16].

More specifically, out of 454 tests (i.e., FOL theories produced by HCC), Z3 successfully checked 327 of them. All of the successful checks were done in less than 0.01s. CVC4 successfully checked 326 out of 454 tested. All of the successful checks were done in less than 0.2s and almost all of those (319) were done in less than 0.015s.

It appears that HCC can only verify statements that are rather simple and could be easily proven by human. In addition, the tool is unpredictable in what it can and cannot verify. The two samples below illustrate this. Both are extracts from the project testsuite.

[2] `https://github.com/danr/contracts.git`

[3] Actually, some statements may be represented by more than one theory. Specifically, statements about recursive values, which are split to be proven by induction.

Example 2.3.1 (Commutativity)**.**

```
(*) `commutativeOver` (===) =
    (*) ::: CF :-> \x ->
            CF :-> \y ->
            CF :&: Pred (=== (y * x))

data Nat = S Nat | Z

(+) :: Nat -> Nat -> Nat
Z   + y = y
S x + y = S (x + y)

max :: Nat -> Nat -> Nat
max Z y         = y
max x Z         = x
max (S x) (S y) = S (max x y)

plus_comm         = (+) `commutativeOver` (==) `Using` eq_refl
max_comm_broken   = max `commutativeOver` (==)
max_comm          = max `commutativeOver` (==) `Using` eq_refl
```

Presented above is an extract from module `Nat.hs`, defining some operations on natural numbers. The obvious definition of `(==)` operator has been omitted. Each of the last three lines defines a statement but only the last one (i.e., `max_comm`) is successfully verified. It asserts that the `max` operator is commutative and hints the tool to use the reflexivity of equality operator (the `eq_relf` statement, also omitted from the extract) in the proof.

A similar statement without any hints as to the proof structure (`max_comm_broken`) could not be checked by either Z3 nor CVC4. A similar statement for the `(+)` operator could also not be verified, even with the hint.

The latter could not be proven by HCC even when provided with a broad collection of hints, namely:

- Reflexivity, symmetry and transitivity of `(==)` operator.
- For all natural numbers `n`: `n + Z == n`.
- For all natural numbers `n` and `m`: `n + (S m) == S (n + m)`.

With the help of Coq proof assistant ([8]) those lemmas were identified as sufficient to make the proof of `plus_comm` statement feasible. Despite that, HCC was unable to produce theory that could be verified by any of the tested provers.

Example 2.3.2 (Risers)**.**

```
import Nat

risers :: [Nat] -> [[Nat]]
risers [] = []
risers [x] = [[x]]
risers (x:y:xs) = case risers (y:xs) of
    s:ss | x <= y    -> (x:s):ss
         | otherwise -> [x]:(s:ss)
    [] -> error "internal error"

risers_cf = risers
    ::: CF :&: Pred (not . null)
    --> CF :&: Pred (not . null)
  `Using` le_cf

risersBy :: (a -> a -> Bool) -> [a] -> [[a]]
risersBy (<) [] = []
risersBy (<) [x] = [[x]]
risersBy (<) (x:y:xs) = case risersBy (<) (y:xs) of
    s:ss | x < y     -> (x:s):ss
         | otherwise -> [x]:(s:ss)
    [] -> error "internal error"

risersBy_cf =
    risersBy ::: (CF --> CF --> CF)
             --> CF :&: Pred (not . null)
             --> CF :&: Pred (not . null)
```

Presented above are definitions of two functions that divide the argument list into sublists, all of which are rising. Function `risers` operates on natural numbers and `risersBy` is its generalization: It accepts any list of any data type and a comparison function as an argument.

There are statements for both functions that assert their crash-freedom and non-emptiness of results, provided that the input lists are not empty. `risers_cf` statement could not be verified by either of the provers, even though it includes a hint to use crash-freedom of the `(<=)` operator. The `risersBy_cf` got successfully verified by CVC4 but not by Z3.

2.3.2. LiquidHaskell

LiquidHaskell repository[4] contains several benchmarks which are annotated (and possibly slightly modified) popular Haskell modules and libraries used in real world applications. The

[4] `https://github.com/ucsd-progsys/liquidhaskell/tree/liquidHaskell-0.1`

most notable are:

- `Data.Map.Base` module.

  The module introduces `Map` data structure and several operations. It provides functionality of a dictionary and is implemented as a balanced binary search tree. It is widely used in Haskell projects.

  Introduced refinements mostly state that:

  1. all the exported functions accept only maps with BST ordering as arguments, and
  2. the ones that produce maps as results preserve this ordering.

  In addition, all the exported functions are verified to terminate.

- `Data.ByteString` library.

  The library provides some functionalities of efficient byte arrays known from imperative languages. To this end, it makes extensive use of basic pointer operations by means of Foreign Function Interface.

  Introduced refinements mostly enforce that exported functions produce `ByteString`s of correct lengths. In addition, some internal functions are refined with predicates related to pointer arithmetic. To make the latter possible, dummy definitions of some foreign functions have been introduced into the source code to allow for reasoning about their behaviour. For example[5]:

```
-- LIQUID foreign import ccall unsafe "string.h memcpy"
   c_memcpy
-- LIQUID      :: Ptr Word8 -> Ptr Word8 -> CSize -> IO (Ptr
    Word8)

{-@ memcpy :: dst:(PtrV Word8)
           -> src:(PtrV Word8)
           -> size: {v:CSize| (v <= (plen src) && v <= (
              plen dst))}
           -> IO ()
  @-}
memcpy :: Ptr Word8 -> Ptr Word8 -> CSize -> IO ()
memcpy p q s = undefined
```

  Most of the functions are, again, verified to terminate.

- `Data.Text` library.

  The library is used for fast and efficient manipulation on Unicode texts. It somewhat resembles `Data.ByteString` in its use of array-type data structures.

  The refinements mostly enforce correct lengths of texts provided to and produced by the functions. In addition, they also put bounds on numerical representation of single characters.

  Again, most functions are verified to be terminating.

File	Time (seconds)
Data/Map/Base.hs	345.7
Data/ByteString.split.0.T.hs	118.9
Data/ByteString.split.0.hs	115.8
Data/ByteString.split.1.T.hs	113.7
Data/ByteString.split.1.hs	108.2
Data/ByteString/Char8.hs	26.9
Data/ByteString/Fusion.hs	63.2
Data/ByteString/Lazy.hs	575.6
Data/ByteString/Unsafe.hs	4.7
Data/ByteString/Fusion.T.hs	74.4
Data/ByteString/Internal.hs	18.1
Data/ByteString/LazyZip.hs	440.8
Data/ByteString/Lazy/Internal.hs	2.5
Data/ByteString/Lazy/Char8.hs	26.8
Total for the library	1690.0
Data/Text.hs	304.0
Data/Text/Internal.hs	6.4
Data/Text/Unsafe.hs	7.5
Data/Text/Search.hs	24.3
Data/Text/UnsafeChar.hs	5.5
Data/Text/Encoding.hs	908.9
Data/Text/Foreign.hs	7.8
Data/Text/Fusion.hs	125.3
Data/Text/Private.hs	2.6
Data/Text/Array.hs	9.3
Data/Text/Lazy.hs	283.7
Data/Text/Fusion/Size.hs	8.5
Data/Text/Lazy/Encoding.hs	15.2
Data/Text/Lazy/Fusion.hs	14.1
Data/Text/Lazy/Builder.hs	40.2
Data/Text/Lazy/Internal.hs	4.7
Data/Text/Lazy/Search.hs	230.0
Total for the library	1998.6

Figure 2.3: LiquidHaskell benchmark results

For the sake of verification process, definitions of many functions were modified. It is hard to estimate, however, how serious those modifications were.

Verification time of those benchmark components on a modern PC was measured for this thesis. Figure 2.3 shows the results. Verification time of a single module can be anything from few seconds to over 10 minutes. Both the `Data.ByteString` and `Data.Text` libraries took about half an hour to verify in total.

On top of that, LiquidHaskell has limited support for incremental verification. That is, it can verify only the parts of source code that have been modified since the last verification. This feature, however, is not further discussed or analyzed, since it is poorly documented and appears to be immature at the current state and require further work.

2.3.3. Conclusions

It appears that HCC is far from having practical applications. It fails to prove non-trivial properties, and may even have difficulties proving the trivial ones. What makes the matter worse, is it does not scale. Providing more resources (CPU time) does not affect performance.

On the other hand, LiquidHaskell appears to be efficient enough for development of real-world projects. If not as a part of standard build procedure, it can be run periodically, e.g. during nightly builds. Benchmarks available in the repository show that the tool is capable of verifying useful refinements within reasonable time constraints for complex software components.

2.4. Case study

The following section aims to provide better intuitive understanding of differences between HCC and LiquidHaskell and their limitations, by describing efforts to verify some operations on popular data structures.

2.4.1. Ordering of maps

Among many tests that come with LiquidHaskell source code, there is an implementation of balanced, ordered functional maps[6]. The following analysis first presents that implementation and concrete refinements being verified, then describes an attempt to verify similar properties in HCC.

LiquidHaskell implementation

Presented below is an extract of the original implementation:

```
{-@
  data Map [mlen] k a < l :: root:k -> k -> Prop
                      , r :: root:k -> k -> Prop>
      = Tip
      | Bin (sz    :: Size)
            (key   :: k)
            (value :: a)
```

[5]Source: `https://github.com/ucsd-progsys/liquidhaskell/blob/liquidHaskell-0.1/benchmarks/bytestring-0.9.2.1/Data/ByteString/Internal.hs`

[6]Source: `https://github.com/ucsd-progsys/liquidhaskell/blob/liquidHaskell-0.1/tests/pos/Map.hs`

```
            (left  :: Map <l, r> (k <l key>) a)
            (right :: Map <l, r> (k <r key>) a)
  @-}

{-@ type OMap k a = Map <{\root v -> v < root}
                        ,{\root v -> v > root}> k a @-}

data Map k a = Tip
             | Bin Size k a (Map k a) (Map k a)

type Size    = Int

{-@ singleton :: k -> a -> OMap k a @-}
singleton :: k -> a -> Map k a
singleton k x
  = Bin 1 k x Tip Tip

{-@ insert :: Ord k => k -> a -> OMap k a -> OMap k a @-}
insert :: Ord k => k -> a -> Map k a -> Map k a
insert kx x t
  = ...
```

It defines a refined type `OMap` that represents any map in BST order. The refinements state that:

- `singleton` produces an ordered map, and
- `insert` may accept only an ordered map as an argument, and
- `insert` produces an ordered map.

No further refinements or annotations are necessary. For this particular implementation, LiquidHaskell infers refined types for all subexpressions, including internal functions, e.g. the ones responsible for keeping the tree balanced.

In addition, a similar refinement for `delete` function is provided in the file.

HCC implementation

For this thesis, parts of this file have been also checked with HCC. However, to make it possible, the data structure was modified to use algebraic natural numbers as keys and size representation:

```
import Nat

data Map a = Tip
           | Bin Size Nat a (Map a) (Map a)

type Size  = Nat
```

This was necessary because HCC does not support typeclasses (and original implementation uses `Ord` typeclass) or non-algebraic types (e.g. `Int`).

Then a predicate was written in Haskell that is true iff a map is ordered:

```
ordered :: Map a -> Bool
ordered = ordered_go Nothing Nothing

ordered_go :: Maybe Nat -> Maybe Nat -> Map t -> Bool
ordered_go _ _ Tip = True
ordered_go bLower bUpper (Bin _ k x l r) =
        (checkBounds bLower bUpper k)
     && (ordered_go bLower (Just k) l)
     && (ordered_go (Just k) bUpper r)

checkBounds :: Maybe Nat -> Maybe Nat -> Nat -> Bool
checkBounds (Just kl) (Just ku) k = kl < k && k < ku
checkBounds (Just kl) Nothing k = kl < k
checkBounds Nothing   (Just ku) k = k < ku
checkBounds Nothing Nothing k = True
```

The `ordered_go` predicate accepts not only map, but also optional bounds. If these are provided, the predicate verifies not only that the map is ordered, but also that all of its keys lie in between those bounds.

Several statements were formed to verify that the predicates are crash-free:

```
ordered_cf = ordered ::: CF --> CF
    `Using` ordered_go_cf

ordered_go_cf = ordered_go ::: CF --> CF --> CF --> CF
    `Using` checkBounds_cf

checkBounds_cf = checkBounds ::: CF --> CF --> CF --> CF
    `Using` lt_cf
```

Several functions were adjusted to the modified definition of `Map` data type. Particularly interesting ones are `singleton`, `singleL` and `singleR`. The latter two define tree rotations (preserving the order) that are used to keep the tree balanced. They use a helper function `bin` that simply glues two subtrees and a key-value pair into a new tree.

```
singleton :: Nat -> a -> Map a
singleton k x = Bin (S Z) k x Tip Tip

singleL :: Nat -> b -> Map b -> Map b -> Map b
singleL k1 x1 t1 (Bin _ k2 x2 t2 t3) =
    bin k2 x2 (bin k1 x1 t1 t2) t3
singleL _  _  _ Tip                  =
    error "sinlgeL Tip"

singleR :: Nat -> b -> Map b -> Map b -> Map b
singleR k1 x1 (Bin _ k2 x2 t1 t2) t3 =
    bin k2 x2 t1 (bin k1 x1 t2 t3)
singleR _  _  Tip _                  =
    error "sinlgeR Tip"

bin :: Nat -> a -> Map a -> Map a -> Map a
```

```
bin k x l r = Bin (size l + size r + (S Z)) k x l r
```

Again, several statements were formed, this time asserting not only crash-freedom, but also ordering of the resulting map:

```
singleton_cf = singleton ::: CF
                         --> CF
                         --> CF :&: (Pred ordered)
    `Using` ordered_cf

singleL_cf = singleL ::: CF
                     :-> (\k -> CF
                     --> CF :&: Pred (ordered_go Nothing (Just k))
                     --> CF :&: Pred (not . empty)
                            :&: Pred (ordered_go (Just k) Nothing)
                     --> CF :&: Pred ordered)
    `Using` ordered_go_cf

singleR_cf = singleR ::: CF
                     :-> (\k -> CF
                     --> CF :&: Pred (not . empty)
                            :&: Pred (ordered_go Nothing (Just k))
                     --> CF :&: Pred (ordered_go (Just k) Nothing)
                     --> CF :&: Pred ordered)
    `Using` ordered_go_cf
```

The contracts for `singleL` and `singleR` functions might look quite complicated. `singleL_cf` states that as long as

- the first map argument to `singleL` is ordered and all its keys are smaller than `k`, and
- the second map argument is ordered and all its keys are greater than `k`, and
- the second map argument is not empty, and
- all the arguments are crash-free

the result is crash-free and ordered.

The contract for `singleR` is similar, only non-emptiness requirement is put on the first, not the second map arguments.

As to the helper `bin` function, two contracts were tried:

```
bin_cf = bin ::: CF
             --> CF
             --> CF
             --> CF
             --> CF
    `Using` plus_cf

bin_ord = bin ::: CF
              :-> (\k -> CF
              --> CF :&: Pred (ordered_go Nothing (Just k))
              --> CF :&: Pred (ordered_go (Just k) Nothing)
```

Statement	Z3	CVC4
`ordered_cf`	✓	✓
`ordered_go_cf`	✓	✓
`checkBounds_cf`	✓	×
`singleton_cf`	✓	×
`bin_cf`	✓	✓
`bin_ord`	✓	×
`singleL_cf ‘Using‘ bin_cf`	✓	×
`singleR_cf ‘Using‘ bin_cf`	×	×
`singleL_cf ‘Using‘ bin_ord`	×	×
`singleR_cf ‘Using‘ bin_ord`	×	×

Figure 2.4: Statements for Map module. ✓ means that statement was successfully verified, × means that the prover exceeded time limit

```
              --> CF :&: Pred ordered)
    ‘Using‘ plus_cf
    ‘Using‘ ordered_go_cf
    ‘Using‘ ordered_cf
```

`bin_cf` simply states crash-freedom. `bin_ord` additionally speaks about ordering and is similar in that regard to `singleL_cf` or `singleR_cf`, only with the non-emptiness requirement dropped. In different runs, statements for `singleL` and `singleR` were provided with hints to use either `bin_cf` or `bin_ord` in the proof.

Results

LiquidHaskell managed to verify that refinements for `singleton`, `insert` and `delete` functions are correct. In several runs, the verification was consistently performed in about 30 seconds in total. The tool inferred refined types for all the internal functions relied upon. For example, the following refined type was inferred for the original `singleL` function:

```
{-@ singleL :: forall a b.
       k1:a
    -> b
    -> OMap {VV : a | (VV < k1)} b
    -> OMap {VV : a | (VV > k1)} b
    -> OMap a b
  @-}
singleL :: a -> b -> Map a b -> Map a b -> Map a b
singleL k1 x1 t1 (Bin _ k2 x2 t2 t3)
    = bin k2 x2 (bin k1 x1 t1 t2) t3
singleL _  _  _ Tip
    = error "sinlgeL Tip"
```

As could be expected, HCC managed to verify only some of the statements. Figure 2.4 shows the results with Z3 and CVC4 used as the back-end theorem provers.

Using CVC4 prover, HCC was unable to prove even that `singleton` produces an ordered map. The results were slightly better when using Z3. Interestingly, correctness could be proven for `singleL` but not for `singleR` even though function definitions and corresponding

contracts are symmetrical. HCC was unable to prove correctness of any of those when hinted to use `bin_ord` statement and not `bin_cf`.

2.4.2. List concatenation

Consider a simple module defining `List` data structure and concatenation operation (`append`):

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

append :: forall a. List a -> List a -> List a
append (Nil) xs = xs
append (Cons x xs) ys = Cons x (xs `append` ys)
```

Some simple function operating on `List` were additionally defined for this thesis and verified using both tools. Presented below are those definitions, specification and verification results.

LiquidHaskell version

Several basic functions were defined together with concrete LiquidHaskell refined type specifications for testing purposes:

```
{-@ id0 :: x:(List a) -> {v:(List a) | x=v} @-}
id0 :: List a -> List a
id0 l = l

{-@ id1 :: x:(List a) -> {v:(List a) | x=v} @-}
id1 :: List a -> List a
id1 Nil = Nil
id1 (Cons x xs) = Cons x xs

{-@ id2 :: x:(List a) -> {v:(List a) | x=v} @-}
id2 :: List a -> List a
id2 Nil = Nil
id2 (Cons x xs) = Cons x (id2 xs)

{-@ addEmpty :: x:(List a) -> {v:(List a) | x=v} @-}
addEmpty x = x `append` Nil

{-@ emptyAdd :: x:(List a) -> {v:(List a) | x=v} @-}
emptyAdd x = Nil `append` x
```

Definitions of `id0`, `id1` and `id2` are increasingly complex but they are in fact identity functions. `addEmpty` and `emptyAdd` are also identity functions since they concatenate the argument with an empty list from the right and left side respectively.

`id0` is easily verified without any modification to the original code. `id1` and `id2`, however, need the following concrete refinements to be provided for data constructors:

```
{-@ Cons :: x:a
         -> xs:(List a)
         -> {v:(List a) | (v = (Cons x xs))} @-}
{-@ Nil :: {v:(List a)| v = Nil} @-}
```

Those make it possible to verify `id1` and `id2` but refinements for `addEmpty` and `emptyAdd` still produce errors. To deal with that, the following refinement for `append` is needed:

```
{-@ append :: x:(List a)
           -> y:(List a)
           -> {v:(List a) | (((y=Nil) => (v=x))
                          && ((x=Nil) => (v=y)))} @-}
```

which in turn requires refinement for `Cons` data constructor to be strengthened by explicitly stating that a value created with `Cons` may not at the same time be a `Nil`:

```
{-@ Cons :: x:a
         -> xs:(List a)
         -> {v:(List a)| ((v = (Cons x xs))
                       && (not (v = Nil)))} @-}
```

HCC version

Since HCC does not support typeclasses, data type again had to be adjusted:

```
data List = Nil | Cons Nat List
```

Otherwise, reasoning about equality of polymorphic lists would require relying on the `Eq` typeclass. Function definitions were adjusted accordingly. Equality for lists was defined:

```
(===) :: List -> List -> Bool
Nil === Nil                   = True
(Cons x xs) === (Cons y ys) = (x == y)
                            && (xs === ys)
```

Several statements were formed to express that list equality is crash-free and that the functions mentioned above are identities. To this end, a helper `isIdentity` function was defined which for a given function `f` produces a statement: *`f` is an identity.*

```
leq_cf = (===) ::: CF
               --> CF
               --> CF
    `Using` eq_cf

isIdentity f = f ::: CF
                 :-> (\x -> CF :&: Pred (=== x))
    `Using` leq_cf

id0_id = isIdentity id0
id1_id = isIdentity id1
id2_id = isIdentity id2
addEmpty_id = isIdentity addEmpty
emptyAdd_id = isIdentity emptyAdd
```

None of these statements could be verified using either Z3 or CVC4. While some of them might seem trivial, it is important to note that for HCC the equality operator `(===)` is just an arbitrary function. For this reason, more statements were added for the operator. Figure 2.5 shows results after hinting the tool to use reflexivity[7] of equality:

[7]Definition of `reflexive` comes from testsuite provided with HCC

Statement	Z3	CVC4
`addEmpty_id`	✗	✓
`emptyAdd_id`	✓	✓
`id0_id`	✓	✓
`id1_id`	✓	✓
`id2_id`	✗	✓
`leq_cf`	✗	✗
`leq_refl`	✗	✗
With symmetry:		
`addEmpty_id`	✓	✗
`emptyAdd_id`	✓	✓
`id0_id`	✓	✓
`id1_id`	✓	✓
`id2_id`	✓	✓
`leq_cf`	✗	✗
`leq_refl`	✗	✗
`leq_symm`	✗	✗

Figure 2.5: Proving identity for lists with HCC. ✓ denotes success, ✗ denotes exceeding the time limit

```
reflexive (~~) =
    All (\x -> x ::: CF :=> x ~~ x ::: CF :&: Pred id)

leq_refl = reflexive (===)

isIdentity f = f ::: CF
                 :-> (\x -> CF :&: Pred (=== x))
    `Using` leq_cf
    `Using` leq_refl
```

Even though reflexivity itself could not be proven, relying on it allowed CVC4 to prove that all of the functions are identities. There was also significant improvement for Z3. Similar statement was introduced that spoke about symmetry of `(===)`. As figure 2.5 shows, it allowed Z3 to succeed with all the functions. Surprisingly, though, CVC4 was unable to finish proof for `addEmpty` with addition of that hint.

Associativity

HCC testsuite contains a general statement expressing associativity for any operation and over any relation:

```
(*) `associativeOver` (===) = All $ \z -> z ::: CF :=>
    (*) ::: CF :-> \x
         -> CF :-> \y
         -> CF :&: Pred (\r -> (r * z) === (x * (y * z)))
```

This can be used to express in HCC associativity of list concatenation:

```
append_comm = append `associativeOver` (===)
    `Using` leq_cf
```

```
    `Using` leq_refl
    `Using` leq_symm
```

Again, the relationship between provided hints and the ability to prove this statements seems complicated. When using only `leq_cf` and `leq_symm`, CVC4 successfully proves it. Adding reflexivity (`leq_refl`) to dependency hints causes it to time out. No set of dependency hints was found that would allow Z3 to successfully prove this statement.

Expressiveness of LiquidHaskell is weaker than that of HCC and thus it is hard to express associativity. There are, however, some hacks possible. One of the ways is defining a function that produces a pair, and annotating it with a refinement stating that the components of the pair are equal:

```
{-@ associative :: x:(List a)
                -> y:(List a)
                -> z:(List a)
                -> (List a, List a)<{\a b -> a=b}> @-}
associative x y z = ( (x `append` y) `append` z
                    ,  x `append` (y `append` z) )
```

The three arguments to `associative` effectively act as universal quantification over possible lists. `append` is associative iff the function `associative` is of the refined type above. However, the issue of proving this refinement remains. In fact, this attempt at working around LiquidHaskell weakness does not come close to solving the problem. It merely delegates it somewhere else. To the best of author's knowledge it is impossible to express and prove associativity of `append` in LiquidHaskell.

2.5. Summary

Neither of the systems can be said to have expressiveness strictly stronger than the other. Nevertheless, informally, it appears that HCC allows for expressing significantly more complex partial correctness properties of a Haskell program than LiquidHaskell. It is of little use, however, since only simple properties seem to be verifiable in practice. What is more, it does not offer any method of dividing proof goal into subgoals that would be sufficient to overcome this limitation. While the achievement of translating Haskell programs to FOL is remarkable and might be a breakthrough in the field of automated verification, it does not seem to have any practical applications at the moment of writing this thesis.

In certain aspects LiquidHaskell seems to be the opposite of HCC. It is fast, predictable and the verification procedure is decidable. Alas, the differences are not limited just to HCC's drawbacks. Expressiveness of LiquidHaskell, unlike that of HCC, is severely limited. The properties it can express, while useful, are quite simple. Describing relationships between different functions (like that between `insert` and `lookup` in `Data.Map`) seems to be beyond its abilities. Despite those limitations, however, it might be beneficial to incorporate it in development of real-world Haskell programs.

2.5.1. Use in teaching

In addition to software development, another potential use for both systems worth discussing is didactics, especially courses in programming languages, functional programming or software verification.

It is difficult, however, to come up with a reasonable way to incorporate HCC in either of those. In its present state, the tool is unpredictable, capable of proving only simple statements

and often can fail even with those. When that happens, it is virtually impossible to identify reason of failure and work around it. Thus, introducing HCC would steepen the learning curve, while providing little to no value. Courses not focusing exclusively on verification can probably benefit more from introducing testing tools such as those mentioned in section 1.3.3, which come with a specification language similar to that of HCC.

LiquidHaskell does not have this issue but there is another problem instead: since its expressiveness is quite limited, it would be a small enhancement of the original course. Thus it is probably best used in the context of various possible type systems, as an interesting working example among others, rather than as a verification tool.

Chapter 3

Verification bugs

At present, usability of both systems is hindered by bugs, some of which are design errors. The following chapter first presents problems found within LiquidHaskell. Then a thorough explanation of a HCC bug follows. It is the bug mentioned in the previous chapter.

3.1. LiquidHaskell

Three kinds of errors have been identified within the LiquidHaskell tool. They are described below, ordered by their significance, starting with the most serious one.

3.1.1. Strict vs. lazy evaluation

LiquidHaskell is based on a similar project for OCaml. While there are many similarities between Haskell and OCaml and programmers proficient with one of those are usually able to quickly learn the other, one difference has a great impact on the project correctness: laziness.

Unlike Haskell, OCaml is strictly evaluated. For this reason refined type inference procedure that works for OCaml programs might provide incorrect results for Haskell.

Example 3.1.1 (Error caused by lazy evaluation)**.** This example has been first published on the LiquidHaskell blog[1]. Consider the following definitions:

```
{-@ divide :: n:Int -> d:{v:Int | v /= 0} -> Int @-}
divide n 0 = error "division by 0"
divide n d = n `div` d

explode = let z = 0
          in  (\x -> (2013 `divide` z)) (foo z)
```

`explode` evaluates to `2013 `divide` 0` and so it crashes. Since Haskell is lazy, the `x` argument and `foo` function will not be evaluated in runtime. LiquidHaskell, however, does not take that into consideration. Instead, it infers their refined types and allows them to affect the refined type of `explode`.

Now, consider the following definition of `foo`:

```
{-@ foo :: n:Int -> {v:Int | 0 <= v && v < n} @-}
foo n | n > 0     = n - 1
      | otherwise = foo n
```

LiquidHaskell verifies that `foo` is of the specified refined type, because that is true whenever `foo` successfully evaluates (as opposed to diverging).

LiquidHaskell further infers that `x` in definition of `explode` is of refined type:

```
{ v:Int | 0 <= v && v < z }
```

from which follows that `0 < z` and so the expression `2013 `divide` z` successfully evaluates[2].

At the time of writing this thesis, LiquidHaskell authors are working to overcome this issue. Their results, however, have not yet been published.

3.1.2. Bounded integer arithmetic

LiquidHaskell formulae language includes basic arithmetic operators which allow for reasoning about integer values. It does not, however, take into consideration bounds of some types, most notably `Int`. Consider the following code sample:

```
{-@ addOne :: x:Int -> { v:Int | v > x } @-}
addOne :: Int -> Int
addOne x = x + 1
```

While at first sight it might appear that `addOne` indeed always produces an integer greater than the argument, it is important to keep in mind that `Int` is a bounded type, represented by a 32-bit or 64-bit word, depending on the architecture. `maxBound` is the maximum possible value and, as could be expected, `addOne maxBound` produces a negative value, even though `maxBound` is positive. Yet, LiquidHaskell incorrectly states that `addOne` is of the refined type provided above.

3.1.3. Minor bugs

There are several minor bugs in LiquidHaskell, mostly dealing with parsing. For example, `Nat` and `List` seem to be treated as reserved keywords, yet when the user tries to use one of them as an identifier for a custom value or data type, LiquidHaskell either provides no error messages or they are confusing and misleading. Such minor problems are to be expected, since there is no formal description of the specification language.

3.2. Haskell Contracts Checker

One might get the impression that there has been more deliberate thought and research put into HCC, compared to LiquidHaskell. And yet, it also did not avoid hidden problems within its verification procedure. Presented below is a concrete example of such problem, followed by a brief analysis of its generalization.

[1] `http://goto.ucsd.edu/~rjhala/liquid/haskell/blog/blog/2013/11/23/telling_lies.lhs/`

[2] This explanation is somewhat simplified but true in essence.

3.2.1. Transitivity

One might notice that in section 2.4.2 list equality operator was explicitly specified as being reflexive and symmetrical, but not transitive, and for a reason: HCC is unable to prove its transitivity. In fact, HCC is unable to prove transitivity of any non-total relation. Consider the following transitivity definition included in HCC testsuite[1]:

```
transitive (~~) = (All $ \x -> All $ \y -> All $ \z ->
    x ::: CF :=> y ::: CF :=> z ::: CF :=>
    x ~~ y ::: CF :&: Pred id :=>
    y ~~ z ::: CF :&: Pred id :=>
    x ~~ z ::: CF :&: Pred id)
```

It is simply a function that, when provided with a binary relation, produces a statement that the relation is transitive (as long as the relation and arguments passed to it are crash-free). Now let's consider this function applied to the list equality operator (definition of which is obvious) and the following two lists:

```
emptyList = Nil
singletonList = Cons Z Nil
```

Those two obviously are not equal. Let them replace `x` and `z` variables in transitivity definition above. We thus obtain the following statement:

```
All $ \y ->
    y ::: CF :=>
    emptyList === y             ::: CF :&: Pred id :=>
    y         === singletonList ::: CF :&: Pred id :=>
    emptyList === singletonList ::: CF :&: Pred id)
```

It reads: *For all crash-free values* `y`*, if equality of* `y` *to both* `emptyList` *and* `singletonList` *is crash-free and true, then equality of* `emptyList` *and* `singletonList` *is also crash-free and true.* Note that the conclusion is within the scope of universal quantification. This statement might appear correct until we recall from section 2.1.5 the purpose and semantics of *unr* constant in translation to FOL.

Let *empty* and *singleton* be the FOL translations of `emptyList` and `singletonList` respectively, and *listEq* be the FOL translation of the `(===)` operator. Let further the `y` variable assume value *unr*. Then the expression:

```
emptyList === y
```

will be translated as:

$listEq(empty, unr)$

and

```
y === singletonList
```

will be translated as:

$listEq(unr, singleton)$

Both those terms ($listEq(empty, unr)$ and $listEq(unr, singleton)$) are equal to *unr* because they rely on the *unr* constant (passed to them as an argument). But the *unr* constant, by definition, meets every contract.

That is, with *unr* substituted for `y`, a SMT prover will verify that the statements:

[1] `https://github.com/danr/contracts/blob/master/testsuite/Properties.hs`

```
emptyList === y              ::: CF :&: Pred id
```

and

```
y          === singletonList ::: CF :&: Pred id
```

are both true. This, according to the transitivity definition above, implies that `emptyList === singletonList`, which obviously is not correct.

3.2.2. Working transitivity definition

The transitivity definition above may be modified to behave as expected:

```
transitive' (~~) = (All $ \x -> All $ \y -> All $ \z ->
    x ::: CF :=> y ::: CF :=> z ::: CF :=>
    not_stmt (unr (x ~~ y)) :=>
    not_stmt (unr (y ~~ z)) :=>
    x ~~ y ::: CF :&: Pred id :=>
    y ~~ z ::: CF :&: Pred id :=>
    x ~~ z ::: CF :&: Pred id)
```

where statement of the form:

```
not_stmt (unr e)
```

asserts that translation of the expression `e` is not equal to *unr*. The `unr` function can be defined as:

```
unr :: u -> Statement u
unr e = e ::: (Pred $ const False)
```

It works because the only object that meets false contract is *unr*. The `not_stmt` can be then defined similarly to how it was defined in section 2.2.2:

```
false_stmt = unr True
not_stmt s = s :=> false_stmt
```

Transitivity defined in such way indeed can be proven true for list equality. It can also be proven false for non-transitive relations. However, the definition of `transitive'` is non-intuitive.

3.2.3. Complex statements

In general, that kind of error may be introduced by using complex statement constructs: implication (`:=>`) and universal quantification (`All`). A programmer using one of those needs to be familiar with HCC's internal workings and take them into consideration. For this reason, at the current state complex statements are probably best avoided, since they can be thought of as expressing properties of a language that has different semantics from Haskell. Using only the basic contract compliance statement (`:::`) allows to avoid such difficulties. This, however, significantly reduces expressiveness of the system.

The cause of this error is clear once one recalls how HCC translates programs into FOL. Most importantly, how it drops type information and represents any ill-typed expression with the *unr* constant, which in turn by definition complies with every contract. The FOL formula representing statement *expression e complies with contract c* is therefore true not only when *e* indeed complies with *c*, but also when *e* is ill-typed. The reason it does not cause errors when using the compliance with a contract statement (`:::`), is because Haskell type checker would

let through only well-typed expressions. When one explicitly uses implication and universal quantification, however, the antecedent is in virtually all cases weaker than it appears.

In general the behaviour and semantics of complex statement constructs are very hard to understand The proof of translation soundness in [16] is limited to contract constructs and the basic, compliance with a contract statement.

Chapter 4

Possible improvements

The following chapter presents possible improvements to both the system. First three improvements expanding LiquidHaskell's functionality are proposed. Then potential ways to work around HCC's bug and poor performance are suggested.

4.1. LiquidHaskell

Of the three enhancements proposed in this section, the first two increase LiquidHaskell's expressiveness. The last one introduces simple notation for what can already be achieved, but with an unnecessarily complex structure.

4.1.1. Termination checking

In the recent years, significant research has been conducted in the field of automatic verification of functional programs termination, detailed summary of which can be found in [12]. Little has been done, however, for lazy higher-order languages such as Haskell. Laziness in particular poses some unique problems, such as infinite data structures.

As mentioned in section 1.2.2, LiquidHaskell has a limited capability of proving program termination. Specifically, it is able to do so for a recursive function that has a specific numeric (or measurable) argument decreasing with each recursive call. Obviously, there are many cases not covered by this functionality. They fall into one or both of the following categories:

- Mutually recursive functions.
- Functions that always terminate, yet don't have single argument that would decrease with each call. For example, the Ackermann function:

```
ackermann :: Integer -> Integer -> Integer
ackermann 0 n = n +1
ackermann m 0 = ackermann (m-1) 1
ackermann m n = ackermann (m-1) (ackermann m (n-1))
```

Both could be addressed by introducing potential functions. I.e., a measure defined not for a data structure, but for a function application. It must be noted that integer-valued measures might not be sufficient for all cases. For example, the Ackermann function presented above requires a pair of integers with lexicographic ordering. That is, potential of the call `ackermann m n` would be the tuple `(m, n)`. A set of mutually recursive functions would

require a separate measure for each of the functions. All of the measures for such set would need to have the same result type.

It appears that LiquidHaskell team is working on a similar functionality at the time of writing this thesis. Yet no solutions have been published so far, neither in a peer-reviewed paper nor in an informal channel.

4.1.2. Abstract refinements within concrete refinements

Whenever verification of map data structure was mentioned in this thesis, it only dealt with ordering. However, to have confidence in the implementation, one must be certain not only that all the operations preserve BST ordering, but also that once a value is inserted at a given key, it can be successfully looked up at that very key, until it is deleted. Given current expressiveness of LiquidHaskell, it is impossible to express and therefore also verify such property. One way to change that, is to allow for using abstract refinements (described in section 1.2.2) from within concrete refinements.

Example 4.1.1 (Map specification with enhanced abstract refinements)**.** Consider the following refined definition of `Map` data structure:

```
{-@
  data Map [mlen] k a < l :: root:k -> k -> Prop
                      , r :: root:k -> k -> Prop
                      , assoc :: k -> a -> Prop>
       = Tip
       | Bin (sz    :: Size)
             (key   :: k)
             (value :: (a<assoc key>))
             (left  :: Map <l, r, assoc> (k <l key>) a)
             (right :: Map <l, r, assoc> (k <r key>) a)
               @-}
```

In addition to the abstract refinements used in previous `Map` definitions, there is `assoc`. The intention behind it is following: `assoc k v` is true only if `k` is a key present in the map and `v` is the value associated with it.

Using it, one might refine `insert` operation as follows:

```
{-@ insert :: forall k a < assoc :: k -> a -> Prop > . Ord k =>
       ik:k ->
       iv:a ->
       Map < {\root v -> v < root }
           , {\root v -> v > root }
           , assoc >
           k a @-}
       Map < {\root v -> v < root }
           , {\root v -> v > root }
           , {\key val -> ((key==ik) => (val==iv)
                        && (key/=ik) => (assoc key val))} >
           k a @-}
```

The first two refinements of the resulting map simply assert its BST order. The last one asserts that the newly inserted key and value are indeed associated with each other, and that associations of all the other keys are intact.

Such a complex refinement might force user to further provide concrete refinements for functions used by `insert`. For example the `singleL` function, previously introduced in section 2.4.1 and used for balancing a tree, might be refined as follows:

```
{-@ singleL :: forall < assocL :: a -> b -> Prop
                      , assocR :: a -> b -> Prop > .
               ik:a ->
               iv:b ->
               Map < {\root v -> v < root}
                   , {\root v -> v > root}
                   , assocL >
                   { v:a | v < ik} b ->
               Map < {\root v -> v < root}
                   , {\root v -> v > root}
                   , assocR >
                   { v:a | v > ik} b ->
               Map < {\root v -> v < root }
                   , {\root v -> v > root}
                   , {\k v -> (assocL k v)
                           || (assocR k v)
                           || (ik==k && iv==v)} >
                   a b @-}
singleL k1 x1 t1 (Bin _ k2 x2 t2 t3) =
    bin k2 x2 (bin k1 x1 t1 t2) t3
```

Refinements such as the one above might appear to put too big a burden on the programmer. However, the certainty and level of trust they provide might be well worth the cost in some applications.

At the current state, refinements such as those in the example are not permitted in LiquidHaskell because they call an abstract refinement from within a concrete refinement, for example towards the end of specification for `singleL`:

```
{\k v -> (assocL k v)
      || (assocR k v)
      || (ik==k && iv==v)}
```

Those, in turn, are not allowed because they would require the fixpoint tool to support liquid variables (see section 2.1.4) used within concrete refinements.

But there is no reason not to allow that, other than the effort required to enhance LiquidHaskell and the fixpoint tool. In fact, LiquidHaskell team is planning to add this capability at some point in the future.

4.1.3. Multiple refinements for a single value

As mentioned in section 2.2.1, LiquidHaskell allows for specifying only one refined type for each value, unlike HCC, which allows unlimited number of statements to be expressed about

single entity. Removing this limitation would allow for a more thorough and readable documentation. It is also possible to easily enhance LiquidHaskell in such manner, since this functionality can be already emulated, as described below. Obviously, all refined types of a single value must share a common underlying sort. This suggestion is thus quite similar to intersection types presented in [6].

Let us assume that the following definition is present in the program:

```
f :: A -> B -> ... -> T
f a b ... = e
```

where `e` is an expression of type `T` that might be using `a`, `b` and other arguments as free variables.

Let there be two concrete refinements that a user wants to specify for function `f`:

```
{-@ f :: a:{ v:A | (P1 v) }
       -> b:{ v:B | (Q1 a v) }
       -> ...
       -> { v:T | (R1 a b ... v) } @-}
{-@ f :: a:{ v:A | (P2 v) }
       -> b:{ v:B | (Q2 a v) }
       -> ...
       -> { v:T | (R2 a b ... v) } @-}
```

where `P1`, `P2`, ... are some concrete predicates.

The user must then force LiquidHaskell to ensure two things:

1. That the function definition belongs to both the refined types.
2. That at each call site at least one of the refined types is respected.

Doing the former is straight-forward. It suffices to give the same definition to another function and then provide each of the copies with only one of the above refinements:

```
{-@ f :: a:{ v:A | (P1 v) }
       -> b:{ v:B | (Q1 a v) }
       -> ...
       -> { v:T | (R1 a b ... v) } @-}
f a b ... = e

{-@ g :: a:{ v:A | (P2 v) }
       -> b:{ v:B | (Q2 a v) }
       -> ...
       -> { v:T | (R2 a b ... v) } @-}
g a b ... = e
```

Enforcing correct behaviour at call sites is slightly more complicated. A new refinement needs to be formulated, that combines the two refinements desired by the user:

```
{-@ f :: a:{ v:A | (P1 v) || (P2 v) }
       -> b:{ v:B | ((P1 a) && (Q1 a v))
                  || ((P2 a) && (Q2 a v)) }
       -> ...
       -> { v:T | ((P1 a) && (Q1 a v) && ... && (R1 a b ... v))
                || ((P2 a) && (Q2 a v) && ... && (R2 a b ... v)) }
```

In short, the resulting predicate refining any argument states: *Among the desired refinements there is at least one that is true for the current argument and all the previous arguments.*

The above method can be trivially generalized to any number of desired refinements. While conducting this task by hand would be burdensome, it can be easily automated. Most importantly, this functionality can be introduced on the parsing level, and thus does not require (possibly complex) adjustments to the verification procedure.

4.2. HCC

Presented below are three possible improvements to HCC. The first one is a suggestion of limitations which, when introduced, make it possible to avoid the verification bug. Following that are briefly described proposals to deal with HCC unpredictability.

4.2.1. Eliminating the verification bug

As explained in section 3.2.1, contract compliance statement (`:::`) in combination with implication (`:=>`) and universal quantification might lead to erroneous behaviour of the contract checker. Presented in this section is a possible way of fixing that problem.

Recall from the section 2.1.5 that a statement of the form:

```
e ::: c
```

is true in any of the three cases:

- when the contract `c` is true for expression `e`.
- when translation of expression `e` is equal to *unr*. That is, `e` either diverges or is ill-typed.
- when translation of contract `c` is equal to *unr*.

The second provision is necessary for arrow contracts (i.e., contracts describing functions), since every arrow contract introduces an implicit universal quantification over all possible arguments. It is not needed, however, for values which are not functions.

Eliminating arrow contracts

Arrow contracts are not necessary element of the syntax. I.e., they can be eliminated without limiting expressiveness of the system. This observation will be used later in this subsection to avoid the verification bug.

Example 4.2.1 (Eliminating arrow contracts)**.** Consider a function of type:

```
f :: a -> b -> c
```

for some types `a`, `b` and `c`. Further consider a statement expressing compliance of that function with a contract of the form:

```
f ::: c1 --> c2 --> c3
```

for some contracts `c1`, `c2` and `c3`. The above statement could be also expressed as:

```
All $ \x -> All $ \y ->
    x ::: c1 :=>
    y ::: c2 :=>
    (f x y) ::: c3
```

That is: *for any two arguments that meet their respective contracts, function `f` applied to those arguments also meets its contract.*

For the sake of clarity, the above example used non-dependent arrow contract and a fixed number of arguments. However, the transformation can be easily generalized to dependent arrow contracts (`:->`) and functions of arbitrary arity. In case of higher-order functions (with nested arrow contracts), the above transformation may be applied iteratively until all such contracts are replaced with equivalent statements.

Transforming arbitrary statements

A compliance with a contract statement is true if the expression or the contract translates to *unr*. Presented below is a way of generalizing this interpretation to any statement. In this new interpretation, an expression translating to *unr* also causes statement to be true, but appropriate checks are separated form the compliance statement (`:::`).

For readability, several new constructs are introduced. None of them increases expressiveness of the system, but they make the following description more easier to follow.

1. Statement conjunction (`:&&:`) and disjunction (`:||:`) operators.
2. A *strict compliance with a contract statement*, denoted by `::::` operator. A statement of the form:

   ```
   e :::: c
   ```

   is true iff the contract `c` is true for expression `e` (and both `e` and `c` successfully evaluate).
3. An *is unreachable* statement, denoted by `UNR(*)` symbol, applicable to expressions. A statement of the form:

   ```
   UNR(e)
   ```

   is true iff translation of `e` is equal to *unr*.

Now let `s` be any statement. Consider the following transformation of `s`:

1. Eliminate all arrow contracts, transforming them as explained above.

2. Assuming there are no quantifiers in implication antecedents, transform the statement to prenex normal form.

3. Maintain set $\mathcal{E}$ of *potentially unreachable expressions* and initialize it to $\emptyset$.

4. For any compliance with a contract statement of the form `e ::: c`, present within statement `s`:

   (a) Replace `e ::: c` with the strict compliance statement `e :::: c`.

   (b) Add `e` to $\mathcal{E}$.

   (c) For any predicate contract of the form `Pred p` present within contract `c`, add `p (e)` to $\mathcal{E}$.

5. Let `f` be the quantifier-free part of the transformed statement `s`. Replace it with:

   ```
   UNR(e_1) :||: UNR(e_2) :||: ... :||: UNR(e_n) :||: (f)
   ```

   where `e_1`, `e_2`, ... `e_n` are all members of the $\mathcal{E}$ set.

Example 4.2.2 (Transforming the transitive statement)**.** Recall the transitivity definition of any binary relation `(~~)`:

```
All $ \x -> All $ \y -> All $ \z ->
    x ::: CF :=> y ::: CF :=> z ::: CF :=>
    x ~~ y ::: CF :&: Pred id :=>
    y ~~ z ::: CF :&: Pred id :=>
    x ~~ z ::: CF :&: Pred id
```

This definition does not contain any arrow contracts and is already in prenex normal form.

After processing all compliance statements, and transforming the quantifier-free part of the statement, the final transitive statement would look as follows:

```
All $ \x -> All $ \y -> All $ \z ->
    UNR(x) :||: UNR(y) :||: UNR(z) :||:
    UNR(x ~~ y) :||: UNR(y ~~ z) :||: UNR(x ~~ z) :||:
    UNR(id (x ~~ y)) :||:
    UNR(id (y ~~ z)) :||:
    UNR(id (x ~~ z)) :||:
    ( x :::: CF :=> y :::: CF :=> z :::: CF :=>
      x ~~ y :::: CF :&: Pred id :=>
      y ~~ z :::: CF :&: Pred id :=>
      x ~~ z :::: CF :&: Pred id )
```

Statements thus transformed explicitly express that the user-provided formula needs to be true only when none of the expressions and contracts translates to *unr*.

Limitation

Alas, the above transformation limits system expressiveness. No existential quantification may be allowed in the final statements, as it would be always trivially true (due to presence of the *unr* constant). For this reason, universal quantifiers cannot be allowed in implication antecedents. Unfortunately, if this restriction is to be posed on all statements, it would severely undermine HCC's usability. Specifically, the tool could be no longer used for verifying higher-order functions.

Consider the following statement for the standard `map` function:

```
map_cf = map ::: (CF --> CF) --> CF --> CF
```

The first iteration of eliminating arrow contracts would result in:

```
map_cf = All $ \f -> All $ \xs ->
    f ::: CF --> CF :=>
    xs ::: CF :=>
    (map f xs) ::: CF
```

The second iteration, performed to eliminate the single remaining arrow contract would result in:

```
map_cf = All $ \f -> All $ \xs ->
    ( All $ \x ->
        x ::: CF :=> (f x) ::: CF
    ) :=>
    xs ::: CF :=>
    (map f xs) ::: CF
```

thus placing a universal quantifier in an implication antecedent.

This issue can be overcome to some extent by transforming statements selectively: There is no need to transform statements that do not use explicit quantification or implication, as the checker can verify them correctly. The transformation need only be performed for more complex statements. Banning existential quantification indeed limits their expressiveness, but in practice it is rarely needed.

Limitations imposed on statements might be also considered an advantage of the presented method, since they are explicit. Refusing to verify a statement is a safer behaviour than accepting it, silently modifying its semantics and producing potentially erroneous result.

It would be beneficial to formally prove soundness of this modification. Even without the proof, however, the method might be considered improvement, since the system already contains unproven and even error-inducing elements, i.e., the implication and universal quantification constructs.

4.2.2. Equality operator

In previous examples of HCC statements often some kind of equality relation was involved (e.g., operator `(==)` in example 2.3.1 or operator `(===)` in section 2.4.2). Theories generated for such statements were hard to prove for both Z3 and CVC4, since the relation was treated by the tools as an arbitrary Haskell function.

All the theories produced by HCC use standard equality ($t_1 = t_2$). In virtually all cases equality relations defined in Haskell programs, when translated, are identical with this FOL equality, restricted to objects representing appropriate type.

Take, for example, equality operator `(==)` defined for type `Nat`:

```
data Nat = S Nat | Z

Z     == Z     = True
(S x) == (S y) = x == y
_     == _     = False
```

For any two values `a` and `b` of type `Nat`, if `a == b`, then their translations a and b are terms that are equal: $a = b$. Similarly, for any two terms t_1 and t_2, if they represent values `t1` and `t2` that are of type `Nat`, then `t1 == t2`.

It might be thus beneficial to introduce a new contract constructor:

```
:=: :: a -> a -> Contract a
```

that expresses equality restricted to a type. There are two arguments supporting this idea:

- Attempts to prove using Coq some of the examples with custom equality relations show that the proof difficulty decreases significantly when using standard equality instead of a custom relation. It is reasonable to assume that SMT provers will show better performance when dealing with ordinary equality that they have been optimized for and equipped with useful lemmas concerning it.
- Such contract would allow for verifying program components by implementing their functionality twice independently, and then stating that both implementations produce the same results. It would also allow for safe program optimization, since programmer might state and verify that the optimized version produces the same results as the original one.

4.2.3. Coq back-end

One of the major difficulties with HCC is the tool's unpredictability. Even though the syntax allows for using lemmas (the `Using` keyword), it is hard to guess what lemmas are needed and at which point provers get stuck. A solution to that might be producing theories not only in SMT 2.0 and TPTP formats, but also in Coq language. With such functionality, whenever all automatic provers fail to prove a theory produced by HCC, users may conduct the proof themselves using Coq to enforce its correctness, or use the proof assistant to identify difficulties, provide more lemmas and re-run HCC with automatic provers. Such functionality has been successfully included in Why3 ([5]), another verification tool.

Another benefit would be identifying obstacles in further HCC development. Experience with proving HCC-produced theories in Coq might provide hints as to which parts of translation engine need more attention or optimization, in order to generate theories that are easier to prove.

Chapter 5

Conclusions

5.1. Summary

In this thesis two tools for verification of Haskell programs have been introduced - Haskell Contracts Checker and LiquidHaskell, both recently published. They have been compared against each other, in regard to their expressiveness and performance. Based on that, their usability in software development and in didactics was assessed.

In its current state, HCC appears to provide little value in both fields. Among its shortcomings, the most important one is unpredictability. The tool often fails to prove even simple statements and reasons for that are unclear.

LiquidHaskell, on the other hand, might be a useful utility in software development. While it cannot be used to fully verify (partial) correctness of a program, due to limited expressiveness, it is capable of identifying some programming errors. It might be also useful in teaching, as an interesting working example of a type system.

There are, however, serious design flaws in both tools, leading to erroneous behaviour. Those bugs have been described, along with an explanation of how they got overlooked. Following that, possible improvements to both systems have been proposed. Suggestions for LiquidHaskell focus on extending its expressiveness and functionality. Suggestions for HCC include enforcing new limitations to avoid the aforementioned bug, and also possible approaches to deal with or work around HCC's unpredictability.

5.2. Comments

It might be disappointing that neither of the tools can be used as a robust, formal verification platform for real-world programs. However, it appears that it's not what their designers had in mind.

LiquidHaskell is an attempt to slightly, yet noticeably strengthen Haskell type system. It succeeds at that, except for the issues with laziness, which are being worked on at the time of writing this thesis. If several minor bugs are dealt with and the syntax is properly documented and further integrated with that of Haskell, it might prove to be valuable addition to the type system. One that provides useful guarantees while putting almost no burden on the user.

HCC developers, on the other hand, appear to have intended their project to be a proof-of-concept, rather than a final product. In addition, it might serve and served as a framework for testing various concepts in Haskell verification, especially various algorithms and strategies of translating Haskell to FOL. Developing a Coq back-end, one of the suggestions from chapter 4, might help HCC serve that purpose, since it would allow to better understand difficulties,

inefficiencies and stumbling blocks within generated theories. It would also allow for proving program correctness interactively.

Lastly, bugs within both projects show importance of basing any verification tools on formal soundness proofs, and also regularly reviewing those proofs whenever new features are added or underlying semantics are changed.

Bibliography

[1] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[2] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[3] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 35(9):268–279, September 2000.

[4] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[5] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, 2013.

[6] Tim Freeman and Frank Pfenning. Refinement types for ml. *SIGPLAN Not.*, 26(6):268–277, May 1991.

[7] Andy Gill and Colin Runciman. Haskell program coverage. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, Haskell '07, pages 1–12, New York, NY, USA, 2007. ACM.

[8] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[9] Neil Mitchell and Colin Runciman. Not all patterns, but enough: An automatic verifier for partial but sufficient pattern matching. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, Haskell '08, pages 49–60, New York, NY, USA, 2008. ACM.

[10] Patrick Rondon. *Liquid Types*. PhD thesis, La Jolla, CA, USA, 2012. AAI3523201.

[11] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. Smallcheck and lazy smallcheck: Automatic exhaustive testing for small values. *SIGPLAN Not.*, 44(2):37–48, September 2008.

[12] Damien Sereni. *Termination Analysis of Higher-Order Functional Programs*. PhD thesis, Oxford University, 2006.

[13] William Sonnex, Sophia Drossopoulou, and Susan Eisenbach. Zeno: A tool for the automatic verification of algebraic properties of functional programs. Technical report, February 2011.

[14] Geoff Sutcliffe. The tptp world - infrastructure for automated reasoning. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 1–12, Berlin, Heidelberg, 2010. Springer-Verlag.

[15] Niki Vazou, Patrick M. Rondon, and Ranjit Jhala. Abstract refinement types. In *Proceedings of the 22Nd European Conference on Programming Languages and Systems*, ESOP'13, pages 209–228, Berlin, Heidelberg, 2013. Springer-Verlag.

[16] Dimitrios Vytiniotis, Simon Peyton Jones, Koen Claessen, and Dan Rosén. Halo: Haskell to logic through denotational semantics. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 431–442, New York, NY, USA, 2013. ACM.