ETH Zurich
Chair of Programming Methodology
Department of Computer Science

Symbolic Execution for Chalice

Master's Thesis

Author:
Malte Schwerhoff
08-932-634

Supervisor:
Dr. Ioannis Kassios
Prof. Dr. Peter Müller

Chair of Programming Methodology
inf Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

11th April 2011

I would like to thank Ioannis Kassios for his committed supervision and for the numerous helpful suggestions made in the course of this thesis, and Prof. Peter Müller for giving me the opportunity to assist in his Formal Methods lecture, without which I might not have chosen such a thesis topic. I would also like to thank Jan Smans for his detailed responses to all my questions about his work, and Simon Hudon and Alexander Summers for various interesting and fruitful discussions that indirectly influenced this thesis.

This thesis is dedicated to my parents and my family for their continuous support and for always letting me know where home is, and to Birte Mowe for bringing sunshine to my heart for six loving years and hopefully many more to come.

Abstract

Symbolic execution has been rediscovered in the last decade as a promising technique for the automated verification of programs and as an alternative to vcg-based approaches, which tend to be fragile in the sense that small changes to a program or its specification can have a significant impact on the verification runtime.

In this thesis we extend the symbolic execution algorithm presented by Smans et al. to Chalice, a Microsoft Research language that uses fractional permissions in order to enable verification of concurrent programs operating on shared mutable data structures and to detect possible deadlocks. Specifications in Chalice are expressed by means of implicit dynamic frames, pure functions and abstract predicates, which allow for a specification language that closely resembles the host language, unlike for example separation logic.

Our algorithm has been implemented in an automated program verifier called Syxc, which has been thoroughly tested and compared with the vcg-based Chalice verifier in order to identify strengths and weaknesses of our symbolic execution algorithm. Syxc has been designed with extensibility and maintainability in mind, and could be used as the basis for a symbolic execution engine for industry-strength programming languages such as Scala.

Contents

List of Figures

List of Listings

List of Tables

1 Introduction

1.1 Overview

This Master's thesis is part of a long-term project dedicated to the development of an automatic program verifier for the Scala programming language [Ode10]. More precisely, the thesis' goal is the implementation and evaluation of a verifier for the Chalice programming language [LM09] based on the concept of *symbolic execution* [Kin76]. The development is expected to enable us to gain first-hand experience with symbolic execution and to compare it to the Chalice verifier, which is based on the concept of *verification condition generation* (vcg). Moreover, the resulting verifier is intended to serve as the basic execution engine for the yet to be developed Scala verifier, should the symbolic execution approach prove successful.

1.2 Symbolic execution

Symbolic execution has been introduced by James King in 1976, and attracted attention in the last decade[1] [BCO05, SJP10] as a possible alternative to vcg-based verification approaches. Verifiers built using the latter approach often tend to be unpredictable in terms of verification time, i.e. small changes to the program or its specification can result in significantly different verification times.

Vcg-based verifiers work by first encoding the complete program in a suitable verification language, e.g. first-order logic, and by then utilising an (automatic) theorem prover to verify that the program's postconditions follow from the program's preconditions and the encoded program itself. In a modular verification setting this usually means that the theorem prover is invoked once for each method, and that it operates on a possibly very complex formula.

Symbolic execution engines, on the other hand, execute every possible branch of a method in succession, collecting only the currently relevant information and forwarding them to a theorem prover in order to verify assertions. As an example, consider an if-then-else statement. Its symbolic execution branches at the guard and yields two subsequent executions, once executing the if-block under the assumption that the guard is true, once executing the else-branch assuming that the guard is false. In general, this approach results in a lot more invocations of the theorem prover, but with significantly smaller formulae to prove. This is assumed to have a positive effect on the verifier's stability and might also facilitate debugging, since it is explicit in which execution branch an assertion fails.

[1] See `http://sites.google.com/site/symexbib/` for a non-exhaustive enumeration of papers on symbolic execution.

1.3 CHALICE

Chalice is an experimental object-oriented programming language developed at Microsoft Research. It supports, among others, fork-join concurrency, exclusive as well as shared access to data, inter-thread communication and static deadlock detection. Chalice uses *implicit dynamic frames* [SJP09a] in combination with *fractional permissions* [Boy03] to determine an upper bound on the set of heap locations that a method might change and to control read- and write-access to possibly shared heap locations. Chalice is publicly available as part of the Boogie verification system[2] [Bar+05] and includes several non-trivial, fully specified examples, e.g. hand-over-hand locking, Owicki-Gries style counters or Peterson's algorithm.

Chalice also includes a vcg-based verifier, which encodes Chalice programs in the Boogie programming language [Lei08] and then uses Boogie to verify the encoding. Boogie implements a classical weakest-precondition calculus and depends itself on an automatic theorem prover, e.g. Z3 [MB08], to discharge the resulting proof obligations. In this report we will refer to the vcg-based Chalice verifier simply as "Chalice" or "the vcg-based (Chalice) verifier".

We have chosen Chalice as the target language of our symbolic-execution-based verifier for several reasons:

- Its set of features is small compared to e.g. Scala, but the supported features are nonetheless interesting and challenging with respect to verification.
- Chalice's specification language is similar to the one used in [SJP10], allowing us to build upon this work and to extend it with fractional permissions, fork-join concurrency and shared mutable data structures.
- The included test suite and the vcg-based verifier facilitate the development of our symbolic-execution-based verifier and enable us to compare the two approaches directly, i.e. on the same language.

1.4 PROCEEDING

The rest of the report is structured as follows: in Section 2 we define the subset of the Chalice programming language that we target with our symbolic execution algorithm, which is presented in Section 3. Syxc, our implementation of the algorithm, is be presented in Section 4. Completeness and runtime performance are discussed in Section 5, where we also compare Syxc to the vcg-based Chalice verifier. Section 6 finally concludes the thesis and enumerates possible future work.

[2] http://boogie.codeplex.com/

2 CHALICE

Despite being a small research language, Chalice is already quite powerful and we cannot give a full introduction to it in the context of this thesis. We therefore encourage the reader to consult the appropriate literature, e.g. [LM09, Lei10].

2.1 INTRODUCTORY EXAMPLES

Listing 1 shows a simple Chalice program alongside its specification and illustrates the concepts of implicit dynamic frames, fractional permissions, pure functions and abstract predicates.

In Chalice, a field is only accessible if the current thread has permissions to do so. Hence, permissions can be used to compute an upper bound to the set of fields a mutator method can modify, thereby *framing* it. Write permissions are unique, i.e. at most one thread can posses them at any given time, and denoted by `acc(c.f)`. Read permissions can be granted to multiple threads, which permits parallel read access to shared data structures, and are denoted by `acc(c.f, `n`)`, where $0 < n < 100$, or `rd(c.f)`. The sum of all read permissions distributed over all threads is at most[1] 100. Shared read access is implemented by means of *fractional permissions*, which theoretically can be distributed among infinitely many threads to allow for parallel computations, and recollected afterwards to regain write access. Where *pure functions* (e.g. getters) abstract over expressions, *predicates* abstract over access assertions. Hence, both constructs are means to implement information hiding.

In Listing 1, the function `get()` is used to hide the field `x`, and the predicate `V` in turn hides access to it. The constructor method requires the caller to have full, i.e. write access to `x`, which the caller gains by instantiating a cell, and it ensures full access to `V`. Since method `inc()` and function `get()` both only require read access to `V`, clients can conclude that `get()` will return the same value after `inc()` has been invoked. We therefore say that `V` frames `get()`.

Listing 2 shows a basic example of how fractional permissions can be used together with fork-join concurrency to implement shared read access. Observe that in a modular verification technique such as Chalice, the assertion would fail if method `square` would require `acc(c.V, 50)`. In that case the current thread would have lost all permissions to `acc(c.V)` after the second fork-statement. Since `get()` is framed by `V` its value is havoced, i.e. we lose the information that `get() == a`. By keeping a small fraction of access to `V` we prohibit that another thread can get full access to `V` and thereby write access to `x`. Thus, we can be sure that `get()` retains its value. Another solution would be to add `ensures c.get() == old(c.get())` to the specifications of method `square`.

[1] It is possible for a method to irrevocably lose permissions by ensuring fewer permissions than have been required, which can be used to implement immutable data structures.

Listing 1: An immutable cell and its specification

```
class ImmutableCell {
  var x: int

  predicate V { acc(x) }

  method ImmutableCell(y: int)
    requires acc(x)
    ensures acc(V) && get() == y
  {
    x := y
    fold V
  }

  function get(): int
    requires rd(V)
  { unfolding rd(V) in x }

  method inc() returns (c: ImmutableCell)
    requires rd(V)
    ensures rd(V) && c != null && acc(c.V)
                  && c.get() == get() + 1
  {
    c := new ImmutableCell
    call c.ImmutableCell(unfolding rd(V) in x + 1)
  }
}
```

Listing 2: Illustrating fork-join concurrency and fractional permissions

```
class Main {
  method run(a: int) {
    var y1: int
    var y2: int
    var c: ImmutableCell := new ImmutableCell
    call c.ImmutableCell(a)
      /* Current thread has acc(c.V, 100) */

    fork tk1 := square(c)
    fork tk2 := square(c)
    // fork tk3 := square(c)
      /* Would fail, not enough permissions are left */
    join y1 := tk1
    join y2 := tk2

    assert y1 * y2 == a * a * a *a
  }

  method square(c: ImmutableCell) returns (y: int)
    requires c != null && acc(c.V, 49)
    ensures c != null && acc(c.V, 49)
                     && y == c.get() * c.get()
  { y := c.get() * c.get() }
}
```

Chalice supports sharing and locking of objects, and it uses so-called *mu-values*, stored in mu-fields, to define an order in which shared objects can be acquired. This *locking order* prevents deadlocks that can occur if multiple threads cyclically wait for each other to release an acquired object [Dij71]. An object can only be acquired if its mu-value is strictly greater than the mu-values of all objects already held, i.e. acquired, by the current thread, which can be tested by an assertion such as `assert waitlevel << x.mu` for a given object `x`. See Listing 3 for an illustrating example.

Mu-fields are read-only fields which can only be modified by share- and unshare-statements. This implies that sharing an object `x` will only succeed if the current thread has gained write access to the corresponding mu-field by e.g. a precondition containing `acc(x.mu)`. After being unshared, an object can be shared again with a different mu-value. That is, the mu-value can change during program execution, just as the holds-status can.

Listing 3: Illustrating shared objects and locking orders

```
class PositiveCell {
  var x: int
  invariant acc(x) && x > 0
}

class Client {
  var c1: PositiveCell
  var c2: PositiveCell

  method sharing()
    requires acc(c1) && acc(c2)
  {
    c1 := new PositiveCell
    c2 := new PositiveCell
    c1.x := 1 /* Establish the monitor invariant ... */
    c2.x := 2 /* ... before sharing the objects */
    share c1 above waitlevel
    share c2 above c1
    assert waitlevel << c1 && c1 << c2
  }

  method sharingFailingInvariant()
    requires acc(c1) && acc(c2)
  {
    c1 := new PositiveCell
    share c1 above waitlevel
      /* Error: monitor invariant might not hold */
  }

  method acquiring()
    requires rd(c1) && c1 != null && rd(c2) && c2 != null
    requires rd(c1.mu) && rd(c2.mu)
    requires waitlevel << c1 && c1 << c2
  {
    acquire c1
    acquire c2
    assert c1.x > 0 && c2.x > 0
    release c1
    release c2
  }

  method acquiringWrongOrder()
    requires rd(c1) && c1 != null && rd(c2) && c2 != null
    requires rd(c1.mu) && rd(c2.mu)
    requires waitlevel << c1 && c1 << c2
  {
    acquire c2
    acquire c1 /* Error: a dead−lock might occur */
  }
}
```

Objects have so-called *monitor invariants*, which contain access permissions to fields of the object and assumptions about them. Sharing an object basically means that the permissions mentioned in the monitor invariant are transferred from the sharing thread into the monitor, from where they are eventually transferred to an acquiring thread. If an object is write-acquired the thread gets exactly those permissions mentioned in the invariant. If the object is read-acquired those permissions are scaled down to only permit read access, as described in Section 3.2.4.

2.2 SYNTAX

Our verifier supports the subset of Chalice defined in Figure 1, which roughly is Chalice without support for inheritance (refinement-relations between classes) and channels. Syxc does not yet implement the semantics of certain features which are syntactically valid, e.g. old-assertions in monitor invariants. See Section 5.3 and Section 5.4 for limitations.

For the sake of simplicity we take the liberty to slightly differ from the real Chalice syntax in the following two points (real syntax mapped to our syntax):

- Access assertions: acc($e_0.f$, e_1) maps to acc($e_0.f$, (e_1, 0)) and rd($e_0.f$, e_1) maps to acc($e_0.f$, (0, e_1)), and likewise for predicate access assertions.
- Lock modes: holds(e) maps to holds(e, W), rd holds(e) maps to holds(e, R), !holds(e) and !rd holds(e) both map to holds(e, N), and analogous for `acquire` and `release`. The latter two, however, may not be used with `N`. See Section 5.3.9 for a limitation implied by this simplification.

In order to be verifiable, we require Chalice programs to be well-formed. Since Syxc uses the existing Chalice parser to create a program's abstract syntax tree (AST), we can assume that the programs are syntactically well-formed (e.g. all mentioned variables, functions and methods are declared) and correctly typed if the parser terminates successfully. The only requirement we add is that assertions have to be *self-framing*, i.e. that they include access assertions for each field and predicate they access.

$$
\begin{array}{lll}
\mathit{program} & ::= & \overline{\mathit{cls}} \\
\mathit{cls} & ::= & \texttt{class } C \texttt{ invariant } \phi\ \{\ \overline{\mathit{mem}}\ \} \\
\mathit{mem} & ::= & \texttt{var } f\colon T \mid \mathit{meth} \mid \mathit{func} \mid \mathit{pred} \\
\mathit{meth} & ::= & \texttt{method m(}\overline{x\colon T}\texttt{) returns (}\overline{y\colon T}\texttt{)} \\
 & & \qquad\qquad \texttt{requires } \phi \texttt{ ensures } \phi \texttt{ lockchange } \overline{e}\ \{\ s\ \} \\
\mathit{func} & ::= & \texttt{function } p\texttt{(}\overline{x\colon T}\texttt{) requires } \phi\ \{\ e\ \} \\
\mathit{pred} & ::= & \texttt{predicate } V\ \{\ \phi\ \} \\
s & ::= & e.f \texttt{ := } e \mid \texttt{var } x\colon T \mid e \texttt{ := } e \mid e \texttt{ := new } C \mid \\
 & & \texttt{if (}e\texttt{) then }\{\ s\ \}\texttt{ else }\{\ s\ \} \mid \texttt{call } \overline{x} \texttt{ := } e.m\texttt{(}\overline{e}\texttt{)} \mid \\
 & & \texttt{fork } x \texttt{ := } e.m\texttt{(}\overline{e}\texttt{)} \mid \texttt{join } \overline{x} \texttt{ := } e \mid \texttt{fold } \mathit{pa} \mid \texttt{unfold } \mathit{pa} \mid \\
 & & \texttt{share } e \texttt{ between } \overline{e} \texttt{ and } \overline{e} \mid \texttt{acquire(}e\texttt{, } \mathit{lm}\texttt{)} \mid \\
 & & \texttt{release(}e\texttt{, } \mathit{lm}\texttt{)} \mid \texttt{unshare } e \mid \texttt{free } e \mid \texttt{assert } \phi \mid \\
 & & \texttt{assume } e \mid \texttt{while (}e\texttt{) invariant } \phi \texttt{ lockchange } \overline{e}\ \{\ s\ \} \\
\phi & ::= & \phi \texttt{ \&\& } \phi \mid e \texttt{ ==> } \phi \mid e \texttt{ ?}\phi\texttt{ :}\phi \mid \mathit{fa} \mid \mathit{pa} \mid e \mid \texttt{old(}\phi\texttt{)} \\
e & ::= & e\ \mathit{op}\ e \mid \texttt{null} \mid \texttt{true} \mid \texttt{false} \mid n \mid \texttt{lockbottom} \mid \\
 & & \texttt{waitlevel} \mid e.f \mid e.p\texttt{(}\overline{e}\texttt{)} \mid \texttt{unfolding } \mathit{pa} \texttt{ in } e \mid \\
 & & \texttt{holds(}e\texttt{, } \mathit{lm}\texttt{)} \mid \texttt{old(}e\texttt{)} \\
\mathit{op} & ::= & \mathit{aop} \mid \mathit{bop} \mid \mathit{rop} \\
\mathit{aop} & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \\
\mathit{bop} & ::= & \texttt{\&\&} \mid \texttt{||} \mid \texttt{!} \mid \texttt{<==>} \mid \texttt{==>} \\
\mathit{rop} & ::= & \texttt{==} \mid \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{<<} \\
\mathit{pa} & ::= & \texttt{acc(}e.V\texttt{, } e\texttt{)} \mid \texttt{rd(}e.V\texttt{, } e\texttt{)} \\
\mathit{fa} & ::= & \texttt{acc(}e.f\texttt{, } e\texttt{)} \mid \texttt{rd(}e.f\texttt{, } e\texttt{)} \\
\mathit{lm} & ::= & \texttt{R} \mid \texttt{W} \mid \texttt{N}
\end{array}
$$

In this figure, overlining represents repetition, n, x, f, m, p, V represent an integer literal, a local variable, a field, a method, a pure function and a predicate, respectively, and T, C represent a type and a class, respectively.

Figure 1: The subset of Chalice supported by Syxc

3

Symbolic execution

This section gradually introduces our symbolic execution algorithm: Sections 3.1 to 3.6 define the background theory of the algorithm. The algorithm itself is presented in sections 3.8 to 3.12 and followed by an illustrating example in sections 3.13. The remaining sections present additional technical details of our algorithm.

3.1 Language

Definition 1. We define the following sets in order to mention Chalice code in our symbolic execution algorithm:

- $\mathcal{X}$, the set of local program variables with typical element x
- $\mathcal{E}$, the set of expressions with typical element e
- Φ, the set of assertions with typical element ϕ
- $\mathcal{S}$, the set of statements with typical element s

3.2 Logic

Definition 2. We define $\mathcal{T}$ to be the *theory* of our symbolic execution algorithm. That is, $\mathcal{T}$ contains all function symbols and corresponding axioms which are necessary for our verification technique. $\mathcal{T}$ will be gradually populated in the following subsections whenever we introduce a new concept.

3.2.1 *Sorts and terms*

Definition 3. Assumptions gained during the symbolic execution of a program are encoded as many-sorted first-order logic terms and formulae. We define

- T, the set of terms with typical element t
- $TA \subset T$, the set of fractional permission terms with typical element $t_\alpha = (n, \epsilon)$

where t can be a first-order term or formula. Most terms and formulae follow directly from Chalice's boolean expressions (see Figure 1), and fractional permission terms directly correspond to fractional permission assertions. Following [SJP10] we use the term *term* to refer to any element of T, i.e. to first-order terms and formulae.

Examples of terms are *null* encoding `null`, $t_x = t_y$ encoding the expression `x == y`, and $C.p(t_v, t, \overline{ts})$ encoding an application of a function $C.p$ with a *snapshot* t_v, a receiver t and a list of arguments $\overline{ts}$.

Definition 4. Each term has a corresponding sort, namely one of *Int*, *Bool*, *Ref*, *Mu*, *LMode* and *Snap*, where *Int* and *Bool* are integers and booleans, respectively, where *Ref* are object references, *Mu* are mu-field values, *LMode* are lock modes and *Snap* are function and predicate snapshots. Snapshots are used to frame predicates and functions, and will be illustrated in Section 3.6. In the rest of the report we will omit sorts whenever the context is unambiguous.

3.2.2 *Pure functions*

For each pure n-ary Chalice function

$$p(x_1 \colon T_1,\ \ldots x_n \colon T_n) \colon T_r$$

declared in a class C there exists a function symbol

$$C.p \colon \mathit{Snap} \times \mathit{Ref} \times S_1 \times \ldots \times S_n \to S_r$$

in $\mathcal{T}$'s signature, where S_i is a sort corresponding to Chalice's type T_i. Note that we do not axiomatise functions globally, i.e. add an axiom relating function applications to the evaluation of their respective bodies, as it is done by the vcg-based verifier. Instead, such a relation is added as yet another term to the path conditions whenever a function application is evaluated. The corresponding rule is presented in Figure 2.

3.2.3 *Locks and locking order*

The logic's signature includes the following built-in functions dealing with locks and their locking order:

$$\begin{array}{l}
\mathit{lockbottom} \colon \mathit{Mu} \\
\mathit{R},\ \mathit{W},\ \mathit{N} \colon \mathit{LMode} \\
\mathit{mu} \colon \mathit{Ref} \times \mathit{Int} \to \mathit{Mu} \\
\mathit{holds} \colon \mathit{Ref} \times \mathit{Int} \to \mathit{LMode} \\
< \colon \mathit{Mu} \times \mathit{Mu} \to \mathit{Bool}
\end{array}$$

lockbottom denotes that an object is not shared at all, *R* and *W* denote a read and a write lock, respectively, and *N* denotes that an object is not locked, i.e. held, by the current thread. *mu* and *holds* encode the current mu-field value and holds-status (*R*, *W* or *N*) of an object. The second argument to *mu* and *holds* intuitively represents the version or revision of the function, with the highest revision being the currently valid one. Such a versioning is necessary in order to support function updates, which will be explained in Section 3.5.

$\mathcal{T}$ also includes the following axioms about locking orders, the *mu* function and the *holds* function:

$$\begin{array}{l}
\forall\, m_1,\ m_2 \bullet m_1 < m_2 \Rightarrow m_2 \neq \mathit{lockbottom} \\
\forall\, m \neq \mathit{lockbottom} \bullet \mathit{lockbottom} < m
\end{array}$$

$$\forall\, m_1,\, m_2 \bullet \neg(m_1 < m_2 \;\wedge\; m_2 < m_1)$$
$$\forall\, m_1,\, m_2,\, m_3 \bullet m_1 < m_2 \;\wedge\; m_2 < m_3 \Rightarrow m_1 < m_3$$
$$\forall\, r_1 \neq r_2,\, v_h,\, v_m \bullet$$
$$\quad \mathit{holds}(r_1,\, v_h) \neq N \;\wedge\; \mathit{holds}(r_2,\, v_h) \neq N \Rightarrow \mathit{mu}(r_1,\, v_m) \neq \mathit{mu}(r_2,\, v_m)$$

The first four axioms define a strict partial order with *lockbottom* as the least element. The fifth axiom declares that two held objects must have distinct mu-values.

3.2.4 *Permissions*

Definition 5. We define a less-than order relation on TA as follows:

$$\forall\, (n_1,\, \epsilon_1),\, (n_2,\, \epsilon_2) \in TA \bullet (n_1,\, \epsilon_1) < (n_2,\, \epsilon_2) \Leftrightarrow n_1 < n_2 \vee (n_1 = n_2 \;\wedge\; \epsilon_1 < \epsilon_2).$$

In addition, we use the comparisons greater-than ($>$), at-least ($\geq$) and at-most ($\leq$) in the context of permission terms and with the usual semantics relative to the definition of less-than. For the sake of conciseness we also define the following shorthand

$$\forall\, p \in \mathit{Int},\, (n,\, \epsilon) \in TA \bullet (n,\, \epsilon) < p \Leftrightarrow n < p \vee \epsilon < 0$$

to conveniently express that a fractional permission is positive ($t_\alpha \geq 0$) or permits write access ($t_\alpha \geq 100$).

Definition 6. Moreover, we define the operations addition, subtraction and the so-called *permission scaling* (multiplication) on TA:

$$\forall\, (n_1,\, \epsilon_1),\, (n_2,\, \epsilon_2) \in TA \bullet (n_1,\, \epsilon_1) \pm (n_2,\, \epsilon_2) = (n_1 \pm n_2,\, \epsilon_1 \pm \epsilon_2)$$
$$\forall\, (n,\, \epsilon) \in TA \bullet (100,\, 0) * (n,\, \epsilon) = (n,\, \epsilon) = (n,\, \epsilon) * (100,\, 0)$$
$$\forall\, (n_1,\, \epsilon_1),\, t_\alpha \in TA \bullet (n_1,\, \epsilon_1) {*} t_\alpha = \begin{cases} (n_1,\, \epsilon_1) & \text{if } t_\alpha = (100,\, 0), \\ (0,\, 1) & \text{if } t_\alpha = (0,\, 1) \;\wedge\; \epsilon_1 = 0 \;\wedge\; n_1 > 0, \\ (0,\, \epsilon_1) & \text{if } t_\alpha = (0,\, 1) \;\wedge\; \epsilon_1 > 0 \end{cases}$$

The intricate definition of scaling is due to the fact that the current fractional permission model in Chalice does not lend itself to generally reversible multiplication. Consequently, Chalice prohibits such operations, namely partial folding and unfolding of predicates containing non-full access permissions. Partial folding and unfolding of predicates containing only full-access permissions is permitted and corresponds to the second axiom in the above definition. In addition, Chalice also permits scaling of monitor invariants by epsilon permissions (and epsilon permissions only!). This is done when an object is read-acquired or -released, and the semantics are captured by the third axiom. For example, read-acquiring a monitor invariant containing `acc(x)` yields `rd(x)` access, as does `acc(x, 50)`, whereas `rd(x,2)` yields `rd(x,2)` access.

3.3 SYMBOLIC STATE

Definition 7. A *symbolic state* is a quadruple $(\gamma, h, g, \pi) \in \Sigma$ composed of a symbolic store γ, a symbolic heap h, a symbolic old heap g and path conditions π, all of which will be defined in the rest of this subsection.

The symbolic state contains all information available while symbolically executing a program. It determines the result of assertion and expression evaluations, and it is modified by assertions and statements. Modifications due to assertions are called *production* and *consumption* of assertions, and presented in Section 3.9 and Section 3.10, respectively. Access to a component $c \in \{\gamma, h, g, \pi\}$ of a heap σ is denoted by an object-oriented style dot-syntax: $\sigma.\gamma, \sigma.h, \sigma.g$ and $\sigma.\pi$. Substituting a heap component c' for a component c yields a new state σ' and is denoted by $\sigma[c'/c]$. Substituting several components is denoted by $\sigma[c_1'/c_1, c_2'/c_2, \dots, c_n'/c_n]$ and defined in a sequential way as $\sigma[c_1'/c_1, c_2'/c_2, \dots, c_n'/c_n] = \sigma[c_1'/c_1][c_2'/c_2] \dots [c_n'/c_n]$.

Definition 8. A *symbolic store* $\gamma \in \Gamma$ is a partial function $\gamma : \mathcal{X} \to T$ from variables to terms. Its domain comprises all variables in the current scope of the symbolic execution.

We denote function updates of γ by $\gamma' = \gamma + (x, t)$, which is defined as

$$\begin{aligned}
&\forall\, \gamma,\ \gamma',\ x,\ t \bullet \\
&\quad \gamma' = \gamma + (x,\ t) \\
&\qquad \Leftrightarrow \\
&\quad \gamma'(x) = t\ \wedge\ \forall\, x' \neq x \bullet \gamma'(x) = \gamma(x)
\end{aligned}$$

Definition 9. A *symbolic heap* $h \in H$ is a set of *heap chunks* representing currently accessible memory cells and their values. Heap chunks are either field chunks, predicate chunks or token chunks, representing accessible fields, predicates and joinable tokens, respectively.

A chunk $c \in CH$ can be added to and removed from a heap h, which is denoted by $h' = h + c$ and $h' = h - c$, respectively, and defined as

$$\begin{aligned}
&\forall\, h,\ h',\ c \bullet h' = h + c \Leftrightarrow h' = h \cup \{c\} \\
&\forall\, h,\ h',\ c \bullet h' = h - c \Leftrightarrow h' = h \setminus \{c\}
\end{aligned}$$

Definition 10. *Field chunks* are of the form $t_r.f \mapsto t_v \;\#\; t_\alpha$, where

- t_r represents a receiver object
- f represents a field of the receiver object
- t_v represents the value of the field (f points to t_v)
- t_α is a fractional permission representing the current thread's access to the field

If such a field chunk exists, then (1) the field $t_r.f$ may at least be read by the current thread and (2) the field's current value is t_v.

Definition 11. *Predicate chunks* are of the form $t_r.V[t_s] \;\#\; t_\alpha$, where

- t_r represents a receiver object
- V represents a predicate of the receiver object
- t_s is a snapshot determine the values of the fields that the predicate V abstracts over, i.e. the heap segments *covered* by V

- t_α is a fractional permission representing the current thread's access to the predicate

As with field chunks, the existence of such a predicate chunk in a heap implies that (1) the corresponding predicate is at least readable by the current thread and (2) that unfolding the predicate yields access to fields whose current value is determined by the predicate's snapshot.

The careful reader might have noticed that field chunks and predicate chunks are structurally identical, unlike the predicate chunks introduced in [SJP10], which also contain the arguments passed to predicates. Parameterised predicates are a feature that is currently not supported by Chalice, but we nevertheless keep the distinction between field and predicate chunks to emphasise the difference in semantics and to facilitate extending our formalism with parameterised predicates.

Definition 12. *Token chunks* are of the form $t_r \mapsto (C.m, h, t, \overline{ts})$, where

- t_r identifies the token itself, i.e. t_r is the term a token variable points to
- $C.m$ represents the forked method
- h is the heap that existed when the method has been forked and in which the monitor invariant held
- t is the receiver object of the invoked method
- $\overline{ts}$ represents the actual method arguments

Definition 13. A set $\pi \in \Pi$ of *path conditions* is a set of first-order logic formulae representing assumptions gained from the symbolic execution in progress. For example, for a given field x (recorded in h) and a local variable y (recorded in γ) with current values t_x and t_y, respectively, an assumption such as `x > y` would go to the path conditions as $t_x > t_y$. Sources of path conditions are, among others, preconditions, postconditions of method calls and guards of if-then-else statements.

Analogous to updating a heap h we update path conditions π by adding a term t, denoted by $\pi' = \pi + t$.

Since two of the four heap components can be uniquely identified by their type and because modifications of the current heap component h are much more common than updates of g, we take the liberty to introduce the following shorthand notations.

$$\begin{aligned}
\sigma + (x,\ t) &= \sigma[(\sigma.\gamma + (x,\ t))\ /\ \gamma]\\
\sigma + c &= \sigma[(\sigma.h + c)\ /\ h]\\
\sigma + t &= \sigma[(\sigma.\pi + t)\ /\ \pi]
\end{aligned}$$

$$\begin{aligned}
\sigma\ /\ \gamma' &= \sigma[\gamma'\ /\ \gamma] &\qquad \sigma\ /\ (x,\ t) &= \sigma[\{(x,\ t)\}\ /\ \gamma]\\
\sigma\ /\ h' &= \sigma[h'\ /\ h] & \sigma\ /\ c &= \sigma[\{c\}\ /\ h]\\
\sigma\ /\ \pi' &= \sigma[\pi'\ /\ \pi] & \sigma\ /\ t &= \sigma[\{t\}\ /\ \pi]
\end{aligned}$$

where $t \in T, x \in \mathcal{X}, c \in CH, \gamma' \in \Gamma, h' \in H, \pi' \in \Pi$.

Definition 14. An atomic term (a variable) t is called *fresh* with respect to a set of states if it is syntactically distinct from all terms in these states. We write $t =$ *fresh* to express that t is such a term, and we use *fresh* as an argument to functions that take a term, or generally in places where a term is expected, e.g. $t_r.f \mapsto$ *fresh* $\#\ t_\alpha$ or $\gamma + (x,\ \textit{fresh})$.

Intuitively[1], the fresh-function corresponds to a function with side-effects that increases a private counter on each invocation and successively returns terms *t1, t2, t3,* etc. The correctness of the rules of our symbolic executing algorithm, for example the production of conjuncted assertions presented in Figure 5, depend on this freshness property.

3.4 INFERRING OBJECT DISTINCTNESS

The fact that the sum of all permissions to a field can be at most 100 allows us to conclude that two objects c_1 and c_2 of the same type C must be distinct if the current thread holds acc($c_1.f$, n_1) and acc($c_2.f$, n_2), where $n_1 + n_2 > 100$. Such a situation is illustrated in Listing 4.

Listing 4: Inferring object distinctness from access permissions

```
method get(n: int, m: int) returns (c1: Cell, c2: Cell)
  requires 0 < n && n <= 100
  requires 0 < m && m <= 100
  ensures c1 != null && c2 != null
  ensures acc(c1.x, n) && acc(c2.x, m)
{ ... }

method aliasingPossible() {
  var c1: Cell
  var c2: Cell
  call c1, c2 := get(50, 50)
  assert c1 != c2
    /* Should fail, aliasing is possible */
}

method aliasingImpossible() {
  var c1: Cell
  var c2: Cell
  call c1, c2 := get(51, 51)
  assert c1 != c2
    /* Should succeed, aliasing is impossible */
}
```

We therefore add the following axiom inferring object distinctness from access permissions, i.e. from field chunks in the current heap, to our theory $\mathcal{T}$:

$$\begin{array}{l} \forall\, t_1,\, t_2,\, t_\alpha,\, t_\beta,\, f \bullet \\ \qquad\quad \exists\, t_1.f \mapsto \_ \,\#\, t_\alpha \in \sigma.h \\ \qquad \land\ \exists\, t_2.f \mapsto \_ \,\#\, t_\beta \in \sigma.h \\ \qquad \land\ t_\alpha + t_\beta > 100 \Rightarrow \\ \qquad\quad t_1 \neq t_2 \end{array}$$

Inequalities established by this axiom also effect function updates of *mu* and *holds*, which will be described in Section 3.5.

[1] Giving a rigorous formal definition of *fresh* is quite challenging and would make it much harder to formalise the symbolic execution rules, and we thus refrain from it.

3.5 FUNCTION UPDATES

Since locking order and hold-status of an object can change during program execution, we need to reflect this when adding terms such as

$$\forall\, r \bullet \textit{holds}(r) \Rightarrow mu(r) < mu(t_x)$$

to the path conditions. That is, we must be able to update functions at certain positions. We achieve this by including two counters v_h and v_m indicating the currently valid revision of *holds* and *mu* as ghost fields in our heap. They are ghost fields in the sense that they cannot be mentioned in a Chalice program and thus cannot be modified by the programmer. This enables us, for example, to update the holds-status for object `x` to 'released', i.e. *N*, by incrementing v_h and adding

$$holds(t_x,\ v_h) = N \,\wedge\, \forall\, r \neq t_x \bullet \textit{holds}(r,\ v_h) = \textit{holds}(r,\ v_h - 1)$$

to the path conditions. Including the counters as field chunks in the heap directly allows for two-state assertions, e.g. `holds(x) == old(holds(x))` or the lockchange-clause.

We use the following shorthand notations to conveniently read and update these counter chunks:

$$\begin{array}{lcl} v = h(v_x) & \Leftrightarrow & \exists\, \textit{thread}.v_x \mapsto v \;\#\; \_ \in h \vee v = 0 \\ h' = h[v_x{+}{+}] & \Leftrightarrow & v = h(v_x) \,\wedge\, h' = h - \textit{fc} + \textit{thread}.v_x \mapsto v + 1 \;\#\; \_ \end{array}$$

Reading a counter has been defined such that non-existing counters have a value of zero. This merely is a convenience relieving us from the necessity of having to explicitly initialise counter chunks whenever we create a new, i.e. empty heap to operate on.

3.6 SNAPSHOTS

Snapshots are used to frame predicates and heap-dependent functions by storing the relevant heap values alongside the predicate or function application term. In this section we will give an illustrating example of how predicate snapshots work, and another one that illustrates how function snapshots work. This should be sufficient to understand the rules of our symbolic execution algorithm presented in Section 3.8 ff. An example that shows how predicate and function snapshots interact will be studied in greater detail in Section 3.13, where we show how snapshots solve an unsoundness existing in the vcg-based Chalice verifier. When illustrating our symbolic execution algorithm we will only show relevant parts of the current symbolic state, e.g. we may omit some or even all path conditions.

3.6.1 *Predicate snapshots*

Listing 5: Illustrating predicate snapshots

```
class Cell {
  var x: int

  predicate V { acc(x) && x >= 0 }

  method Cell(a: int)
    requires a >= 0 && acc(x)
    ensures V
  {
    x := a
      // $\gamma : \mathit{this} \mapsto t, a \mapsto t_a$
      // $h : t.x \mapsto t_a$
      // $\pi : t_a \geq 0$
    fold V
      // $h : t.V[t_a]$
  }

  method set(a: int)
    requires V
    ensures V
  {
      // $\gamma : \mathit{this} \mapsto t, a \mapsto t_a$
      // $h : t.V[t_v]$
    unfold V
      // $h : t.x \mapsto t_v$
      // $\pi : t_v \geq 0$
    assert x >= 0
    x := a * a
      // $h : t.x \mapsto t_a * t_a$
    fold V
      // $h : t.V[t_a * t_a]$
  }
}
```

When V is folded in method Cell in Listing 5, the snapshot of the resulting predicate chunk is created from the access assertions that are part of the predicate body. Intuitively, folding takes away access permissions from the current thread and stores them together with the corresponding heap values in a predicate chunk. The opposite happens in method set: unfolding the initial predicate chunk sets the value of x to the chunk's snapshot value, about which we only know that it is at least zero. Folding V after updating x recreates the predicate chunk, but with a different snapshot.

3.6.2 *Function snapshots*

Function snapshots are like predicate snapshots, except that they are created from a function's precondition, not from its body. Listing 6 illustrates function snapshots and shows how snapshots are used to create function application terms that are, in a sense, versioned by their snapshots.

Listing 6: Illustrating function snapshots

```
class Cell {
  var x: int

  function get(): int
    requires rd(x)
  { -x }

  method useGet(a: int)
    requires acc(x)
  {
    var y: int
    var z: int

    x := a
      // $\gamma : this \mapsto t, a \mapsto t_a, y \mapsto t_y, z \mapsto t_z$
      // $h : t.x \mapsto t_a$
    y := get()
      // $\gamma : y \mapsto Cell.get(t_a, t), z \mapsto t_z$
      // $\pi : Cell.get(t_a, t) = -t_a$
    x := a - 1
      // $h : t.x \mapsto t_a - 1$
    z := get()
      // $\gamma : y \mapsto Cell.get(t_a, t), z \mapsto Cell.get(t_a - 1, t)$
      // $\pi : Cell.get(t_a, t) = -t_a, Cell.get(t_a - 1, t) = 1 - t_a$
    assert get() != y
  }
}
```

3.6.3 *Logic*

The logic's signature includes the following built-in functions dealing with snapshots:

unit: *Snap*
combine: *Snap* × *Snap* → *Snap*

unit is the empty snapshot used for heap-independent predicates and functions, and *combine* is used to build snapshots from conjuncted assertions. The latter is axiomatised by

$$\forall\, t_1,\ t_2,\ t_3,\ t_4 \bullet \mathit{combine}(t_1,\ t_2) = \mathit{combine}(t_3,\ t_4) \Rightarrow t_1 = t_3\ \wedge\ t_2 = t_4,$$

which is included in our theory $\mathcal{T}$.

3.7 TECHNICAL PREFACE

In the symbolic execution rules presented in the following subsections we use $\sigma \vdash \phi$ to denote that a formula ϕ follows from a state σ. This notation actually is a shorthand for $\mathcal{T}(\sigma.h) \cup \sigma.\pi \vdash \phi$, which states that ϕ follows from our background theory $\mathcal{T}$ together with the current path conditions. $\mathcal{T}(\sigma.h)$ denotes that some elements of $\mathcal{T}$ depend on the current heap, namely the object distinctness axiom added to $\mathcal{T}$ in Section 3.4.

Following [SJP10] we present our rules in a *continuation-passing style* (CPS) [FW08]. In CPS, sequential execution is achieved by passing the remaining computation Q as an argument to the first computation f to perform, i.e. $f(\ldots, Q)$. f invokes Q after its own computation has been performed, thereby continuing with the remaining computation. We consider most of the rules to be self-explaining and usually do not give additional descriptions.

We use [ ] to denote the empty list, $\overline{es}$ to denote a list of arbitrary size, $[e_1, \ldots, e_n]$ to denote a list with n elements, $e :: \overline{es}$ to denote prepending element e to list $\overline{es}$ and $\overline{es_1} ::: \overline{es_2}$ to denote concatenation of lists $\overline{es_1}$ and $\overline{es_2}$.

3.8 EVALUATING EXPRESSIONS

Chalice expressions are evaluated into terms, which may be added to π as further assumptions, or which have to be asserted or refuted in order for the verification to succeed. The evaluation of an expression can itself add path conditions to π – as done in the case of function application evaluation – and thus can have an effect on the state. However, all other state components are guaranteed to remain unchanged.

Expressions are evaluated by one of the following functions:

- eval : $\Sigma \times \mathcal{E} \times (\Sigma \times T \to \mathit{Bool}) \to \mathit{Bool}$, evaluating an expression into a term.
- evals : $\Sigma \times \overline{\mathcal{E}} \times (\Sigma \times \overline{T} \to \mathit{Bool}) \to \mathit{Bool}$, evaluating a list of expressions.

eval and evals are defined in terms of the following helper functions:

- eval′ : $\Sigma \times \Pi \times \overline{\mathcal{F}} \times \mathcal{E} \times (\Pi \times T \to \mathit{Bool}) \to \mathit{Bool}$,

  where the second argument accumulates path conditions yielded by the ongoing evaluation, and where the third argument represents a list of functions currently being evaluated, which is necessary in order to avoid an infinite recursive descent when evaluating recursive functions. See Section 5.3.6 for a related incompleteness of the vcg-based verifier that does not exist in Syxc.
- evals′ : $\Sigma \times \Pi \times \overline{\mathcal{E}} \times \overline{T} \times (\Pi \times \overline{T} \to \mathit{Bool}) \to \mathit{Bool}$,

  where the second argument accumulates path conditions and where the fourth argument accumulates terms yielded by the ongoing evaluation.

The evaluation of function applications and unfolding-expressions is defined in terms of produce and consume which are described in subsequent section. Basically, consume succeeds if a given assertion holds, and consuming access assertions corresponds to losing them. The opposite holds for produce: boolean assertions are produced by adding corresponding assumptions to the path conditions, and access assertions are produced by adding corresponding chunks to the heap.

The rules for the symbolic evaluation of expressions are presented in Figure 2 and Figure 3.

```
$\mathsf{eval}(\sigma,\ e,\ Q) =$
  $\mathsf{eval}'(\sigma,\ \emptyset,\ [\,],\ e,\ (\lambda\,\pi,\ t\,\bullet$
    $Q(\sigma + \pi,\ t)))$

$\mathsf{eval}'(\sigma,\ \pi,\ rs,\ \texttt{c},\ Q) =$
  $Q(\pi,\ t_c)$
  where $\texttt{c}$ is a Chalice literal and $t_c$ is the corresponding term literal.

$\mathsf{eval}'(\sigma,\ \pi,\ rs,\ x,\ Q) =$
  $Q(\pi,\ \sigma.\gamma(x))$

$\mathsf{eval}'(\sigma,\ \pi,\ rs,\ \texttt{old(}e\texttt{)},\ Q) =$
  $\mathsf{eval}'(\sigma[g\ /\ h],\ \pi,\ rs,\ e,\ Q)$

$\mathsf{eval}'(\sigma,\ \pi,\ rs,\ \texttt{!}e,\ Q) =$
  $\mathsf{eval}'(\sigma,\ \pi,\ rs,\ e,\ (\lambda\,\pi1,\ t\,\bullet$
      $Q(\pi1,\ \neg t)))$

$\mathsf{eval}'(\sigma,\ \pi,\ rs,\ e1 \oplus e2,\ Q) =$
  $\mathsf{eval}'(\sigma,\ \pi,\ rs,\ e1,\ (\lambda\,\pi1,\ t1\,\bullet$
    $\mathsf{eval}'(\sigma,\ \pi1,\ rs,\ e2,\ (\lambda\,\pi2,\ t2\,\bullet$
      $Q(\pi2,\ t1 \oplus' t2)))))$
  where $e_1,\ e_2 \neq \texttt{waitlevel}$, $\oplus \in \{$&&, ||, ==>, <==>, ==, <, <=, >, >=, +, -, *, / , %, <<$\}$,
  and where $\oplus'$ is a corresponding operator defined on terms.
```

Figure 2: Symbolic evaluation of expressions

- Evaluating an if-then-else expression can branch the symbolic execution. The if-branch is evaluated if it cannot be proven that the guard will evaluate to false, and the continuation is then invoked assuming the guard is true. Likewise, the else-branch is evaluated if it cannot be proven that the guard evaluates to true.
- A field access is valid if the receiver is not null and if the current thread has permissions to read that field, i.e. if there is a matching field chunk in the heap.
- Evaluating a function application succeeds if the precondition holds. If so, a function application term is created and related to the evaluated function body. In order to avoid an infinite recursion, the body is evaluated only once. Notice that the current rule does not consider information hiding, because the function body is always evaluated and thus visible to all clients. This, however, is not an innate property of our technique, and Chalice in fact already supports the concept of modules to define visibility scopes.
- Unfolding a predicate in an expression temporarily gains the assertion hidden by the predicate and evaluates the expression in the resulting state. Unfolding's effect on the heap is implicitly reverted and hence only temporarily, because evaluation does not pass the current heap to the continuation. Again, information hiding is not yet considered and predicates can be unfolded regardless of the current visibility scope.

```
$\mathsf{eval}'(\sigma,\ \pi,\ rs,\ (n,\ \epsilon),\ Q) =$
  $\mathsf{eval}'(\sigma,\ \pi,\ rs,\ n,\ (\lambda\, \pi 1,\ tn \bullet$
    $\mathsf{eval}'(\sigma,\ \pi 1,\ rs,\ \epsilon,\ (\lambda\, \pi 2,\ t\epsilon \bullet$
      $Q(\sigma + \pi 2,\ (tn,\ t\epsilon))))))$

$\mathsf{eval}'(\sigma,\ \pi,\ rs,\ e0\ \texttt{?}\ e1\ \texttt{:}\ e2,\ Q) =$
  $\mathsf{eval}'(\sigma,\ \pi,\ rs,\ e0,\ (\lambda\, \pi 1,\ t0 \bullet$
    $(\sigma \nvdash \neg t0 \Rightarrow \mathsf{eval}'(\sigma,\ \pi 1 + t0,\ rs,\ e1,\ Q)) \wedge$
    $(\sigma \nvdash t0 \quad \Rightarrow \mathsf{eval}'(\sigma,\ \pi 1 + \neg t0,\ rs,\ e2,\ Q))))$

$\mathsf{eval}'(\sigma,\ \pi,\ rs,\ e\,\texttt{.}f,\ Q) =$
  $\mathsf{eval}'(\sigma,\ \pi,\ rs,\ e,\ (\lambda\, \pi 1,\ t0 \bullet$
    $\sigma \vdash t0 \neq \textbf{\textit{null}} \ \wedge\ \exists\, t0.f \mapsto tv\ \#\ \_ \in \sigma.h \wedge$
      $Q(\pi 1,\ tv)))$

$\mathsf{eval}'(\sigma,\ \pi,\ rs,\ e0\,\texttt{.}p(e1,\ \ldots,\ en),\ Q) =$
  $\mathsf{eval}'(\sigma,\ \pi,\ rs,\ e0,\ (\lambda\, \pi 1,\ t0 \bullet$
    $\mathsf{evals}'(\sigma,\ \pi 1,\ [e1,\ \ldots,\ en],\ [\,],\ (\lambda\, \pi 2,\ ts \bullet$
      $\sigma + \pi 2 \vdash t0 \neq \textbf{\textit{null}} \wedge$
        $\mathsf{let}\ \sigma 1 = \sigma[\{(\textbf{\textit{this}},\ t0),\ (x1,\ ts_1),\ \ldots,\ (xn,\ ts_n)\}\ /\ \gamma] + \pi 2\ \mathsf{in}$
          $\mathsf{consume}(\sigma 2,\ \textbf{\textit{full}},\ pre_{C.p},\ (\lambda\, \sigma 3,\ tv \bullet$
            $\mathsf{let}\ \mathit{tf} = C.p(tv,\ t0,\ ts)\mathsf{in}$
              $\mathsf{if}\ C.p \notin rs$
                $\mathsf{eval}'(\sigma 2,\ \sigma 3.\pi,\ C.p\ ::\ rs,\ body_{C.p},\ (\lambda\, \pi 4,\ tb \bullet$
                  $Q(\pi 4 + (\mathit{tf} = tb),\ \mathit{tf})))$
              $\mathsf{else}$
                $Q(\sigma 3.\pi,\ \mathit{tf})))))))$

$\mathsf{eval}'(\sigma,\ \pi,\ rs,\ \texttt{unfolding acc(}e0\texttt{.}P\texttt{,}\ \alpha\texttt{) in}\ e1,\ Q) =$
  $\mathsf{eval}'(\sigma,\ \pi,\ rs,\ e0,\ (\lambda\, \pi 1,\ t0 \bullet$
    $\mathsf{eval}'(\sigma,\ \pi 1,\ rs,\ \alpha,\ (\lambda\, \pi 2,\ t\alpha \bullet$
      $\sigma + \pi 2 \vdash t0 \neq \textbf{\textit{null}} \ \wedge\ t\alpha \geq 0 \wedge$
        $\mathsf{consume}(\sigma + \pi 2,\ \textbf{\textit{full}},\ \texttt{acc(}e0\texttt{.}P\texttt{,}\ \alpha\texttt{)},\ (\lambda\, \sigma 1,\ tv \bullet$
          $\mathsf{produce}(\sigma 1[(\textbf{\textit{this}},\ t0)\ /\ \gamma],\ tv,\ t\alpha,\ body_{C.P},\ (\lambda\, \sigma 2 \bullet$
            $\mathsf{eval}'(\sigma 2[\sigma.g\ /\ g,\ \sigma.\gamma\ /\ \gamma,\ \emptyset\ /\ \pi],\ \sigma 2.\pi,\ rs,\ e1,\ Q)))))))))$

$\mathsf{eval}'(\sigma,\ \pi,\ rs,\ \texttt{holds(}e\texttt{,}\ lm\texttt{)},\ Q) =$
  $\mathsf{eval}'(\sigma,\ \pi,\ rs,\ e,\ (\lambda\, \pi 1,\ t \bullet$
    $\mathsf{eval}'(\sigma,\ \pi 1,\ rs,\ m,\ (\lambda\, \pi 2,\ tlm \bullet$
      $Q(\pi 2,\ \textbf{\textit{holds}}(t,\ \sigma.h(v_h))\ =\ tlm))))$

$\mathsf{eval}'(\sigma,\ \pi,\ rs,\ \texttt{waitlevel} \oplus e\texttt{.mu},\ Q) =$
  $\mathsf{eval}'(\sigma,\ \pi,\ rs,\ e\texttt{.mu},\ (\lambda\, \pi 1,\ t \bullet$
    $Q(\pi 1,\ \forall\, o \bullet \textbf{\textit{holds}}(o,\ \sigma.h(v_h))\ \neq\ N \Rightarrow \textbf{\textit{mu}}(o,\ \sigma.h(v_m))\ \oplus'\ t)))$
  where $\oplus$ is either << or == and $\oplus'$ is the corresponding operator defined on terms.
```

Figure 2: Symbolic evaluation of expressions (continued)

$$\begin{array}{l} \mathsf{evals}(\sigma,\ es,\ Q) = \\ \quad \mathsf{evals}'(\sigma,\ \emptyset,\ es,\ [\,],\ (\lambda\,\pi,\ ts \bullet Q(\sigma + \pi,\ ts))) \end{array}$$

$$\begin{array}{l} \mathsf{evals}'(\sigma,\ \pi,\ [\,],\ ts,\ Q) = \\ \quad Q(\sigma + \pi,\ ts) \end{array}$$

$$\begin{array}{l} \mathsf{evals}'(\sigma,\ \pi,\ e\ ::\ \ es,\ ts,\ Q) = \\ \quad \mathsf{eval}'(\sigma,\ \pi,\ [\,],\ e,\ (\lambda\,\pi1,\ t \bullet \\ \qquad \mathsf{evals}'(\sigma,\ \pi1,\ es,\ t\ ::\ \ ts,\ Q))) \end{array}$$

Figure 3: Symbolic evaluation of a list of expressions

3.9 PRODUCING ASSERTIONS

Producing assertions corresponds to the concept of *inhaling* [LM09] used in the vcg-based Chalice verifier. Basically, the assertion is evaluated and the resulting term is used as an additional assumption during the rest of the execution. Producing an assertion can effect the current heap and the path conditions, but the other state components will remain unaltered.

Assertions are produced by the function

$$\mathsf{produce}\colon \Sigma \times T \times TA \times \Phi \times (\Sigma \to \mathit{Bool}) \to \mathit{Bool}$$

where the second argument is the snapshot term used to produce the assertion, and where the third argument is a factor used to scale access permissions that are to be produced. produce is defined in terms of the helper function

$$\mathsf{produce}'\colon \Sigma \times T \times TA \times \Phi \times (H \times \Pi \to \mathit{Bool}) \to \mathit{Bool}$$

which passes only the newly produced heap chunks and path conditions to the continuation.

In order to ensure that assertions are self-framing, the production is started in a state where the heap contains only the ghost counter field chunks introduced in Section 3.5. Hence, field reads and other heap-dependent operations will fail if the assertion currently being produced does not include corresponding access assertions. More specifically, access permissions must precede corresponding field reads in the assertion. That is, the order of the conjuncts forming an assertion is of relevance, in contrast to regular truth-valued formulae. The heap resulting from a successful production is afterwards merged with the initial heap. Due to fractional permissions both heaps may contain field chunks (or predicate chunks, respectively) with the same receiver and field but with different values and different fractional permissions. The existence of such a chunk in the initial heap implies that the current thread already has read access to the field, which in turn implies that the field's value cannot be changed by other threads. The new chunk's value must therefore be equal to the initial chunk's value and the new chunk's permission are added to those already held by the thread.

The rules for the symbolic production of assertions are presented in Figure 5.

- Producing a conjunction takes place by producing both conjuncts with their own fresh snapshot term and by refining the initial snapshot to be equal

```
produce($\sigma,\ ts,\ t\alpha,\ \phi,\ Q$) =
  let
    $h_v = \{\mathit{thread}.v_h \mapsto \_ \# \_,\ \mathit{thread}.v_m \mapsto \_ \# \_\} \subseteq \sigma.h,$
    $h0 = \sigma.h \setminus h_v$
  in
    produce'($\sigma \mathbin{/} h_v,\ ts,\ t\alpha,\ \phi,\ (\lambda\, h1,\ \pi 1 \bullet$
      let $(hr,\ \pi r)$ = merge($h0,\ h1$) in
        $Q(\sigma \mathbin{/} hr \mathbin{/} (\pi 1 + \pi r))))$

merge($h1,\ h2$) =
  foldl($h2,\ (h1,\ \emptyset),\ (\lambda\, (h,\ \pi),\ c \bullet$
    if $c = t.f \mapsto tv \# t\alpha \ \wedge\ \exists\, fc @ t.f \mapsto tw \# t\beta \in h$
      $(h - fc$
        $+\ t.f \mapsto tv \# (t\alpha + t\beta),$
      $\pi + (tv = tw))$
    else if $c = t.V[tv] \# t\alpha \ \wedge\ \exists\, pc @ t.V[tw] \# t\beta \in h$
      $(h - pc$
        $+\ t.V[tv] \# (t\alpha + t\beta),$
      $\pi + (tv = tw))))$
```

where foldl is the left-folding higher-order function as e.g. available in Haskell with the first argument being the data structure to iterate over, the second argument being the initial accumulator and the third argument being the combining function

Figure 4: Defining assertion production in terms of helper functions

to the pair of these two fresh snapshots. Using fresh snapshots to produce the conjuncts implements the separating semantics of assertion conjunctions in a very strict manner, such that `acc(x) && acc(y)` results in two disjoint heap chunks. The task of preventing that e.g. `acc(x, 10) && acc(x, 10)` also results in two disjoint heap chunks is delegated to the production rule for access assertions and implemented by means of the already defined merge-operation.

- A field access assertion is produced by merging a corresponding field chunk with the heap produced so far. In case of producing a mu-field access assertion a corresponding term is added to the path conditions. Producing predicate access assertions is handled analogously.

```
produce$'(\sigma,\ tv,\ t\alpha,\ \texttt{true},\ Q) = Q(\sigma.h,\ \sigma.\pi)$
produce$'(\sigma,\ tv,\ t\alpha,\ \texttt{false},\ Q) = \textit{true}$

produce$'(\sigma,\ tv,\ t\alpha,\ \phi1\ \texttt{\&\&}\ \phi2,\ Q) =$
  let $tv1 =$ $\textit{fresh}$, $tv2 =$ $\textit{fresh}$ in
    produce$'(\sigma + (tv = \textit{Combine}(tv1,\ tv2)),\ tv1,\ t\alpha,\ \phi1,\ (\lambda\, h1,\ \pi1 \bullet$
      produce$'(\sigma1\ /\ h1\ /\ \pi1,\ tv2,\ t\alpha,\ \phi2,\ Q)))$

produce$'(\sigma,\ tv,\ t\alpha,\ e\ \texttt{==>}\ \phi,\ Q) =$
  eval$(\sigma,\ e,\ (\lambda\, \sigma1,\ t \bullet$
    $(\sigma1 \nvdash \neg t \Rightarrow$ produce$'(\sigma1 + t,\ tv,\ t\alpha,\ \phi,\ Q)) \wedge$
    $(\sigma1 \nvdash t \quad \Rightarrow Q(\sigma1.h,\ \sigma1.\pi + \neg t))))$

produce$'(\sigma,\ tv,\ t\alpha,\ e0\ \texttt{?}\ \phi1\ \texttt{:}\ \phi2,\ Q) =$
  eval$(\sigma,\ e0,\ (\lambda\, \sigma1,\ t0 \bullet$
    $(\sigma1 \nvdash \neg t0 \Rightarrow$ produce$'(\sigma1 + t0,\ tv,\ t\alpha,\ Q)) \wedge$
    $(\sigma1 \nvdash t0 \quad \Rightarrow$ produce$'(\sigma1 + \neg t0,\ tv,\ t\alpha,\ \phi2,\ Q))))$

produce$'(\sigma,\ tv,\ t\alpha,\ \texttt{acc(}e.f\texttt{,}\ \alpha\texttt{)},\ Q) =$
  eval$(\sigma,\ e,\ (\lambda\, \sigma1,\ t \bullet$
    eval$(\sigma1,\ \alpha,\ (\lambda\, \sigma2,\ t\alpha1 \bullet$
      $\sigma2 \vdash t \neq \textit{null}\ \wedge\ t\alpha1 > 0\ \wedge\ t\alpha1 \leq 100 \wedge$
        let
          $fc = t.f \mapsto tv\ \#\ (t\alpha * t\alpha1)$
          $t\mu =$ if $f = \texttt{mu}$ then $\textit{mu}(t,\ \sigma.h(v_m))\ =\ tv$ else $\textit{true}$,
          $h3,\ \pi3 =$ merge$(\sigma2.h,\ \{fc\})$
        in
          $Q(h3,\ \sigma2.\pi + \pi3 + t\mu)))))$

produce$'(\sigma,\ tv,\ t\alpha,\ \texttt{acc(}e.V\texttt{,}\ \alpha\texttt{)},\ Q) =$
  eval$(\sigma,\ e,\ (\lambda\, \sigma1,\ t \bullet$
    eval$(\sigma1,\ \alpha,\ (\lambda\, \sigma2,\ t\alpha1 \bullet$
      $\sigma2 \vdash t \neq \textit{null}\ \wedge\ t\alpha1 > 0\ \wedge\ t\alpha1 \leq 100 \wedge$
      let $h3,\ \pi3 =$ merge$(\sigma2.h,\ \{t.V[tv]\ \#\ (t\alpha * t\alpha1)\})$ in
        $Q(h3,\ \sigma2.\pi + \pi3)))))$

produce$'(\sigma,\ tv,\ t\alpha,\ \texttt{holds(}e\texttt{,}\ m\texttt{)},\ Q) =$
  eval$(\sigma,\ e,\ (\lambda\, \sigma1,\ t \bullet$
    eval$(\sigma1,\ m,\ (\lambda\, \sigma2,\ tm \bullet$
      let $v = \sigma2.h(v_h) + 1,\ h3 = \sigma2.h[v_h{+}{+}]$in
        $Q(h3,\ \sigma2.\pi + (\textit{holds}(t,\ v)\ =\ tm)$
                     $+ (\forall o \neq\ t \cdot \textit{holds}(\ t,\ v) = \textit{holds}(\ t,\ v - 1)))))))$
```

We assume that negated holds-assertions, i.e. `!holds(`e`, _)`, are represented as `holds(`e`, N)`. See Section 5.3.9 for resulting limitations of this approach.

```
produce$'(\sigma,\ tv,\ t\alpha,\ \texttt{lockchange}\ \overline{es},\ Q) =$
  evals$(\sigma,\ \overline{es},\ (\lambda\, \sigma1,\ \overline{ts} \bullet$
    $Q(\sigma1.h,\ \sigma1.\pi + (\forall o \in \overline{ts} \cdot \textit{holds}(o,\ \sigma1.h(v_h)) = \textit{holds}(o,\ \sigma1.g(v_h))))))$

produce$'(\sigma,\ tv,\ t\alpha,\ e,\ Q) =$
  eval$(\sigma,\ e,\ (\lambda\, \sigma1,\ t \bullet$
    $Q(\sigma1.h,\ \sigma1.\pi + t)))$
```

Figure 5: Symbolic production of assertions

3.10 CONSUMING ASSERTIONS

Consuming assertions corresponds to the concept of *exhaling* used in the vcg-based Chalice verifier. Depending on the concrete assertion the consumption proceeds by proving that the assertion actually holds, or by removing chunks from the heap. In case of the latter the snapshot of the removed chunk is returned. Therefore, consuming an assertion can effect the current heap and the path conditions, but the other state components will remain unaltered.

Assertions are consumed by the function

$$\mathsf{consume}\colon \Sigma \times TA \times \Phi \times (\Sigma \times T \to Bool) \to Bool$$

where the second argument TA is a factor used to scale the access permissions that are to be consumed. consume is defined in terms of the helper function

$$\mathsf{consume}'\colon \Sigma \times H \times TA \times \Phi \times (H \times \Pi \times T \to Bool) \to Bool$$

which passes remaining heap chunks, newly gained path conditions and the consumed snapshot to the continuation. The additional heap argument (second argument) represents the remaining heap which can be modified, i.e. pruned, and which is eventually passed to the continuation. The state heap (first argument) is used to evaluate expressions in. This separation is necessary in order to verify the call-site of a method such as

```
method m()
  requires acc(x) && x == 0
{...}
```

where consuming the second conjunct would fail because the consumption of the first conjunct has already removed the required field chunk.

The rules for the symbolic consumption of assertions are presented in Figure 6. Consuming a field access assertion is successful if the current thread holds the required access permissions, where the latter must be positive and not greater than the permissions currently held. A distinction is made between completely and partially losing permissions to a field. In the latter case the corresponding field chunk is updated to reflect the remaining permissions. The former case includes a dedicated handling of mu-fields by invalidating (havocing) the entry corresponding to the receiving object in the *mu*-function. In both cases the field's value is passed to the continuation as the consumed snapshot.

```
consume$(\sigma,\ t\alpha,\ \phi,\ Q) =$
  consume$'(\sigma,\ \sigma.H,\ t\alpha,\ \phi,\ (\lambda\, h1,\ \pi1,\ tv \bullet$
    $Q(\sigma\ /\ h1\ /\ \pi1,\ tv)))$

consume$'(\sigma,\ h,\ t\alpha,\ \phi1$ && $\phi2,\ Q) =$
  consume$'(\sigma,\ h,\ t\alpha,\ \phi1,\ (\lambda\, h1,\ \pi1,\ tv1 \bullet$
    consume$'(\sigma\ /\ \pi1,\ h1,\ t\alpha,\ \phi2,\ (\lambda\, h2,\ \pi2,\ tv2 \bullet$
      $Q(h2,\ \pi2,\ \mathit{Combine}(tv1,\ tv2))))))$

consume$'(\sigma,\ h,\ t\alpha,\ e$ ==> $\phi,\ Q) =$
  eval$(\sigma,\ e,\ (\lambda\, \sigma1,\ t \bullet$
    $(\sigma1 \nvdash \neg t \Rightarrow$ consume$'(\sigma1 + t,\ h,\ t\alpha,\ \phi,\ Q)) \wedge$
    $(\sigma1 \nvdash t \quad \Rightarrow Q(h,\ \sigma.\pi + \neg t,\ \mathit{unit}))))$

consume$'(\sigma,\ h,\ t\alpha,\ e0$ ? $\phi1$ : $\phi2,\ Q) =$
  eval$(\sigma,\ e0,\ (\lambda\, \sigma1,\ t0 \bullet$
    $(\sigma1 \nvdash \neg t0 \Rightarrow$ consume$'(\sigma1 + t0,\ h,\ t\alpha,\ \phi1,\ Q)) \wedge$
    $(\sigma1 \nvdash t0 \quad \Rightarrow$ consume$'(\sigma1 + \neg t0,\ h,\ t\alpha,\ \phi2,\ Q))))$

consume$'(\sigma,\ h,\ t\alpha,$ acc($e.f$, $\alpha$)$,\ Q) =$
  eval$(\sigma,\ e,\ (\lambda\, \sigma1,\ t \bullet$
    eval$(\sigma1,\ \alpha,\ (\lambda\, \sigma2,\ t\alpha0 \bullet$
      let $t\alpha2 = t\alpha * t\alpha0$ in
            $\sigma2 \vdash t \neq \mathit{null}$
          $\wedge\ \exists\, \mathit{fc} \mathbin{@} t.f \mapsto tv \mathbin{\#} t\alpha1 \in h$
          $\wedge\ \sigma2 \vdash t\alpha2 > 0\ \wedge\ t\alpha2 \leq t\alpha1\ \wedge$
          if $\sigma2 \vdash t\alpha2 = t\alpha1$
            let $(h3,\ t\mu) =$
              if $f =$ mu then
                let $v = h(v_m){+}1$ in $(h[v_m{+}{+}],\ \forall o \neq\ t{\cdot}\mathit{mu}(\,t,\ v) = \mathit{mu}(\,t,\ v{-}1))$

              else
                $(h,\ \mathit{true})$
            in
              $Q(h3 - \mathit{fc},\ \sigma2.\pi + t\mu,\ tv)$
          else
            $Q(h - \mathit{fc} + t.f \mapsto tv \mathbin{\#} t\alpha1 - t\alpha2,\ \sigma2.\pi,\ tv)))))$

consume$'(\sigma,\ h,\ t\alpha,$ acc($e.V$, $\alpha$)$,\ Q) =$
  eval$(\sigma,\ e,\ (\lambda\, \sigma1,\ t \bullet$
    eval$(\sigma1,\ \alpha,\ (\lambda\, \sigma2,\ t\alpha0 \bullet$
      let $t\alpha2 = t\alpha * t\alpha0$ in
          $\sigma2 \vdash t \neq \mathit{null}$
        $\wedge\ \exists\, \mathit{pc} \mathbin{@} t.V[tv] \mathbin{\#} t\alpha1 \in h\ \wedge$
        $\wedge\ \sigma2 \vdash t\alpha2 > 0\ \wedge\ t\alpha2 \leq t\alpha1)\ \wedge$
          if $\sigma2 \vdash t\alpha2 = t\alpha1$
            $Q(h - \mathit{pc},\ \sigma2.\pi,\ tv)$
          else
            $Q(h - \mathit{pc} + t.V[tv] \mathbin{\#} t\alpha1 - t\alpha2,\ \sigma2.\pi,\ tv)))))$

consume$'(\sigma,\ h,\ t\alpha,$ lockchange $\overline{es},\ Q) =$
  evals$(\sigma,\ \overline{es},\ (\lambda\, \sigma1,\ \overline{ts} \bullet$
    $\sigma1 \vdash \forall o \in \overline{ts} \cdot \mathit{holds}(o,\ \sigma1.h(v_h)) = \mathit{holds}(o,\ \sigma1.g(v_h))\ \wedge$
      $Q(h,\ \sigma1.\pi,\ \mathit{unit})$

consume$'(\sigma,\ h,\ t\alpha,\ e,\ Q) =$
  eval$(\sigma,\ e,\ (\lambda\, \sigma1,\ t \bullet$
    $\sigma1 \vdash t\ \wedge$
      $Q(h,\ \sigma1.\pi + t,\ \mathit{unit})))$
```

Figure 6: Symbolic consumption of assertions (continued)

3.11 Executing statements

The three functions introduced so far - eval, produce and consume - all handle assertions or expressions, but do not cover statements and are thus not suited to actually execute programs.

Statements are executed by the functions

- exec : $\Sigma \times \mathcal{S} \times (\Sigma \rightarrow \mathit{Bool}) \rightarrow \mathit{Bool}$
- execs : $\Sigma \times \overline{\mathcal{S}} \times (\Sigma \rightarrow \mathit{Bool}) \rightarrow \mathit{Bool}$

executing a single and multiple statements, respectively. The execution functions are implemented in terms of eval, produce and consume, and they may effect the whole state, i.e. all of its four components.

The rules for the symbolic execution of statements are presented in Figure 7.

- Assert-statements are executed by temporarily consuming the assertion in order to verify that it holds. In contrast, assume-statements are not executed by producing the assumption but by evaluating it. The reason for this is that assumptions would need to be self-framing if they are to be produced. That is, given a field `x` the statement `assume x > 0` would fail and only `assume rd(x) && x > 0` would succeed. But that would also increase the fractional permissions the current thread holds, a consequence that is probably undesired in general. Choosing eval over produce, though, limits assumptions to expressions and thereby disallows assuming permissions.
- Objects are instantiated by creating a corresponding entry in the store and adding corresponding field chunks to the current heap. Note that fields are not initialised to default values such as `0`, `false` or `null`, and that the newly instantiated object can only be assigned to a local variable, not to a field. The latter is not crucial for our approach, but facilitates keeping the rules small and simple.
- The invocation of a method basically corresponds to the consumption of its precondition followed by the production of its postcondition and lockchange clause in the appropriate environments, i.e. stores and heaps. Following [Lei10] we could have encoded call-statements in terms of fork- and join-statements. However, we did not do so, assuming that a dedicated rule for call-statements might facilitate the understanding of the rules for fork- and join-statements.
- Executing a fork-statement succeeds if the method's precondition holds, in which case a token chunk will be added to the heap. The chunk contains the heap in which the precondition held, the receiver object and the actual method arguments. The latter two are necessary in order to restore the call-site environment when eventually joining the token. Re-evaluating the expressions denoting receiver object and actual method arguments at join-time is not an option, because their values might have changed between fork and join. Since tokens can be reused in another fork without joining the previous one first, we have to remove possibly existing heap chunks related to the token before invoking the continuation.
- Sharing an object between lower and upper bounds requires the object to not be shared already and the bounds to be valid, i.e. all lower bounds have to be smaller than all upper bounds. The shared object's mu-field is updated

$$\mathsf{execs}(\sigma,\ [\,],\ Q) = Q(\sigma)$$

$$\begin{aligned}&\mathsf{execs}(\sigma,\ s\ ::\ ss,\ Q) =\\&\quad \mathsf{exec}(\sigma,\ s,\ (\lambda\,\sigma 1 \bullet\\&\qquad \mathsf{execs}(\sigma 1,\ ss,\ Q)))\end{aligned}$$

$$\begin{aligned}&\mathsf{exec}(\sigma,\ \texttt{assert}\ \phi,\ Q) =\\&\quad \mathsf{consume}(\sigma,\ \mathit{full},\ \phi,\ (\lambda\,\sigma 1,\ \_\bullet\\&\qquad Q(\sigma\ /\ \sigma 1.\pi)))\end{aligned}$$

$$\begin{aligned}&\mathsf{exec}(\sigma,\ \texttt{assume}\ e,\ Q) =\\&\quad \mathsf{eval}(\sigma,\ e,\ (\lambda\,\sigma 1,\ t \bullet\\&\qquad Q(\sigma 1 + t)))\end{aligned}$$

$$\begin{aligned}&\mathsf{exec}(\sigma,\ \texttt{var}\ x\!: C,\ Q) =\\&\quad Q(\sigma + (x,\ \ \mathit{fresh}))\end{aligned}$$

$$\begin{aligned}&\mathsf{exec}(\sigma,\ x\ \texttt{:=}\ e,\ Q) =\\&\quad \mathsf{eval}(\sigma,\ e,\ (\lambda\,\sigma 1,\ t \bullet\\&\qquad Q(\sigma 1 + (x,\ t))))\end{aligned}$$

Figure 7: Symbolic execution of statements

to reflect that the object is now shared and the object is declared to not be held. The order in which the steps 1) update the mu-version, 2) assume that the object is shared with respect to the bounds and 3) consume the monitor invariant are executed is crucial for the soundness of the share-statement. Consuming the invariant before sharing the object might lead to an inconsistent monitor invariant, as illustrated in Listing 7. The example verifies in Syxc and in Chalice since the object is shared between `waitlevel` and `c` *before* consuming the monitor invariant. Changing this order would allow us to incorrectly verify the example with the modified monitor invariant `rd(mu) && mu == lockbottom`, leading to a contradiction when the object is acquired.

- The execution of a while loop comprises two steps, the verification of the loop body and the actual execution. The first conjunct of the rule verifies the loop body and includes well-formedness checks, the second conjunct verifies the call-site. The rule is optimised with respect to the handling of local variables, in the sense that local variables declared prior to the loop and not assigned to in the loop's body retain their values, i.e. are not havoced. The operations involved in the rule as well as their order give rise to the incompleteness described in Section 5.3.2.

Listing 7: A snippet illustrating the importance of the order of operations when executing a share-statement

```
class Test {
  invariant rd(mu) && mu != lockbottom

  method init()
    requires acc(mu) && mu == lockbottom
  { share this above waitlevel }
}
```

```
$\mathsf{exec}(\sigma,\ x\ \texttt{:=}\ \texttt{new}\ C,\ Q) =$
  $\mathsf{let}\ t =\ \mathit{fresh},\ t\mu = \mathit{bottomlock}\ \mathsf{in}$
    $Q(\sigma + (x,\ t)$
        $+ \{t.f_1 \mapsto\ \mathit{fresh} \mathbin{\#} \mathit{full},\ \ldots,\ t.f_n \mapsto\ \mathit{fresh} \mathbin{\#} \mathit{full},\ t.\texttt{mu} \mapsto t\mu \mathbin{\#} \mathit{full}\}$
        $+ (t \neq \mathit{null}) + \mathit{mu}(t,\ \sigma.h(v_m))\ =\ t\mu)$
  where $f_1$ to $f_n$ are all fields of class $C$ except $\texttt{mu}$.

$\mathsf{exec}(\sigma,\ e0.f\ \texttt{:=}\ e1,\ Q) =$
  $\mathsf{eval}(\sigma,\ e0,\ (\lambda\, \sigma1,\ t0 \bullet$
    $\sigma1 \vdash t0 \neq \mathit{null}\ \wedge\ \exists\ \mathit{fc} \mathbin{@} t0.f \mapsto \_ \mathbin{\#} t\alpha \in \sigma1.h\ \wedge\ \sigma1 \vdash t\alpha \geq 100 \wedge$
      $\mathsf{eval}(\sigma1,\ e1,\ (\lambda\, \sigma2,\ t1 \bullet$
        $Q(\sigma2 - \mathit{fc} + t0.f \mapsto t1 \mathbin{\#} t\alpha)))))$

$\mathsf{exec}(\sigma,\ \texttt{if}\ e\ \texttt{then}\ s1\ \texttt{else}\ s2,\ Q) =$
  $\mathsf{eval}(\sigma,\ e,\ (\sigma1,\ t \bullet$
    $(\sigma1 \nvdash \neg t \Rightarrow \mathsf{exec}(\sigma1 + t,\ s1,\ Q)) \wedge$
    $(\sigma1 \nvdash t \quad \Rightarrow \mathsf{exec}(\sigma1 + \neg t,\ s2,\ Q)))$

$\mathsf{exec}(\sigma,\ \texttt{call}\ r1,\ \ldots,\ rm\ \texttt{:=}\ e.m\texttt{(}ex1,\ \ldots,\ exn\texttt{)},\ Q) =$
  $\mathsf{eval}(\sigma,\ e,\ (\lambda\, \sigma1,\ t \bullet$
    $\sigma1 \vdash t \neq \mathit{null} \wedge$
      $\mathsf{evals}(\sigma1,\ [ex1,\ \ldots,\ exn],\ (\lambda\, \sigma2,\ [tx1,\ \ldots,\ txn] \bullet$
        $\mathsf{let}$
          $x\gamma = \{(x1,\ tx1),\ \ldots,\ (xn,\ txn)\},$
          $y\gamma = \{(y1,\ \ \mathit{fresh}),\ \ldots,\ (ym,\ \ \mathit{fresh})\}$
        $\mathsf{in}$
          $\mathsf{consume}(\sigma2\ /\ x\gamma + (\mathit{this},\ t),\ \mathit{full},\ \mathit{pre}_{C.m},\ (\lambda\, \sigma3,\ \_ \bullet$
            $\mathsf{produce}(\sigma3[\sigma2.h\ /\ g]{+}y\gamma,\ \ \mathit{fresh},\ \mathit{full},\ \mathit{post}_{C.m}\ \texttt{\&\&}\ \mathit{lkch}_{C.m},\ (\lambda\, \sigma4 \bullet$

              $Q(\sigma4[\sigma2.\gamma\ /\ \gamma,\ \sigma2.g\ /\ g]$
                    $+ \{(r1,\ \sigma4.\gamma(y1)),\ \ldots,\ (rm,\ \sigma4.\gamma(ym))\})))))))))$
  where $\mathit{pre}_{C.m}$, $\mathit{post}_{C.m}$ and $\mathit{lkch}_{C.m}$ are the precondition, postcondition and lockchange clause,
  respectively, of the method to be invoked.

$\mathsf{exec}(\sigma,\ \texttt{fork}\ ek\ \texttt{:=}\ e.m\texttt{(}ex1,\ \ldots,\ exn\texttt{)},\ Q) =$
  $\mathsf{eval}(\sigma,\ e,\ (\lambda\, \sigma1,\ t \bullet$
    $\mathsf{eval}(\sigma1,\ ek,\ (\lambda\, \sigma2,\ tk \bullet$
      $\sigma2 \vdash t \neq \mathit{null} \wedge$
        $\mathsf{evals}(\sigma2,\ [ex1,\ \ldots,\ exn],\ (\lambda\, \sigma3,\ [tx1,\ \ldots,\ txn] \bullet$
          $\mathsf{let}\ x\gamma = \{(x1,\ tx1),\ \ldots,\ (xn,\ txn)\}\ \mathsf{in}$
            $\mathsf{consume}(\sigma3\ /\ x\gamma + (\mathit{this},\ t),\ \mathit{full},\ \mathit{pre}_{C.m},\ (\lambda\, \sigma4,\ \_ \bullet$
              $Q(\sigma4[\sigma3.\gamma\ /\ \gamma,\ \sigma3.g\ /\ g]$
                    $-\ tk.\mathit{joinable} \mapsto \_ \mathbin{\#} \_$
                    $+\ tk.\mathit{joinable} \mapsto \mathit{true} \mathbin{\#} \mathit{eps}$
                    $-\ tk \mapsto (\_, \_, \_, \_)$
                    $+\ tk \mapsto (C.m, \sigma2.h, t, [tx1,\ \ldots,\ txn]))))))))))$
```

Figure 7: Symbolic execution of statements (continued)

```
exec($\sigma$, join $r1, \dots, rm$ := $ek$, $Q$) =
  eval($\sigma$, $ek$, ($\lambda\, \sigma1,\ tk \bullet$
        $\sigma1 \vdash tk \neq \mathit{null}$
      $\wedge\ \exists\, fc\ @\ tk.\mathit{joinable} \mapsto \mathit{true} \mathbin{\#} t\alpha \in \sigma1.h$
      $\wedge\ \exists\, tc\ @\ tk \mapsto (C.m, g1, t, [tx1,\ \dots,\ txn]) \in \sigma1.h\ \wedge$
       let
         $x\gamma = \{(x1,\ tx1),\ \dots,\ (xn,\ txn)\}$,
         $y\gamma = \{(y1,\ \mathit{fresh}),\ \dots,\ (ym,\ \mathit{fresh})\}$,
         $\sigma2 = \sigma1[g1\ /\ g]\ /\ x\gamma + y\gamma + (\mathit{this},\ t)$
       in
         produce($\sigma2$, $\mathit{fresh}$, $\mathit{full}$, $post_{C.m}$ && $lkch_{C.m}$, ($\lambda\, \sigma3 \bullet$
           evals($\sigma3$, $lkch_{C.m}$, ($\lambda\, \sigma4,\ [t1,\ \dots,\ tl] \bullet$
             $Q(\sigma4[\sigma1.g\ /\ g,\ \sigma1.\gamma\ /\ \gamma]$
                 $-\ fc - tc$
                 $+\ \{(r1,\ \sigma4.\gamma(y1)),\ \dots,\ (rm,\ \sigma4.\gamma(ym))\}$
                 $+\ tk.\mathit{joinable} \mapsto \mathit{false} \mathbin{\#} t\alpha)))))))$

exec($\sigma$, fold acc($e.V$, $\alpha$), $Q$) =
  eval($\sigma$, $e$, ($\lambda\, \sigma1,\ t \bullet$
    eval($\sigma1$, $\alpha$, ($\lambda\, \sigma2,\ t\alpha \bullet$
      $\sigma2 \vdash t \neq \mathit{null}\ \wedge\ t\alpha \geq 0\ \wedge$
        consume($\sigma2\ /\ (\mathit{this},\ t)$, $t\alpha$, $body_{C.V}$, ($\lambda\, \sigma3,\ tv \bullet$
          let $(hr,\ \pi r)$ = merge($\sigma3.h$, $\{t.V[tv] \mathbin{\#} t\alpha\}$) in
            $Q(\sigma3\ /\ hr + \pi r)))))))$
```

where *bodyC.V* is the predicate's body.

```
exec($\sigma$, unfold acc($e.V$, $\alpha$), $Q$) =
  eval($\sigma$, $e$, ($\lambda\, \sigma1,\ t \bullet$
    eval($\sigma1$, $\alpha$, ($\lambda\, \sigma2,\ t\alpha \bullet$
      $\sigma2 \vdash t \neq \mathit{null}\ \wedge\ t\alpha \geq 0\ \wedge$
        consume($\sigma2$, $\mathit{full}$, acc($e.V$, $\alpha$), ($\lambda\, \sigma3,\ tv \bullet$
          produce($\sigma3\ /\ (\mathit{this},\ t)$, $tv$, $t\alpha$, $body_{C.V}$, ($\lambda\, \sigma4 \bullet$
            $Q(\sigma4\ /\ \sigma2.\gamma)))))))))$

exec($\sigma$, share $e$ between $\overline{el}$ and $\overline{eu}$, $Q$) =
  eval($\sigma$, $e$, ($\lambda\, \sigma1,\ t \bullet$
        $\sigma1 \vdash t \neq \mathit{null}$
      $\wedge\ \exists\, fc\ @\ t.\mathit{mu} \mapsto \mathit{lockbottom} \mathbin{\#} t\alpha \in \sigma1.h$
      $\wedge\ \sigma1 \vdash t\alpha \geq 100\ \wedge$
       let
         $pre = el_1$ << $eu_1$ && ... && $el_1$ << $eu_m$ && ... && $el_n$ << $eu_m$,
         $post = el_1$ << $e$ && ... && $el_n$ << $e$ && $e$ << $eu_1$ && ... && $e$ << $eu_m$
       in
         consume($\sigma1$, $\mathit{full}$, $pre$, ($\lambda\, \sigma2,\ \_ \bullet$
           let
             $t\mu = \mathit{fresh}$,
             $v = \sigma2.h(v_m) + 1$,
             $\sigma3 = \sigma2\ /\ \sigma2.h[v_m{+}{+}]$
                     $+\ \forall o \neq\ t \cdot \mathit{mu}(\ t,\ v) = \mathit{mu}(\ t,\ v - 1)$
                     $+\ \mathit{mu}(t,\ v)\ =\ t\mu + (t\mu \neq lockbottom)$
                     $+\ \mathit{holds}(t,\ \sigma4.h(v_h))\ =\ N$
                     $-\ fc + t.\mathit{mu} \mapsto t\mu \mathbin{\#} t\alpha$
           in
             eval($\sigma3$, $post$, ($\lambda\, \sigma4,\ tw \bullet$
               consume($\sigma4\ /\ (\mathit{this},\ t) + tw$, $\mathit{full}$, $inv_C$, ($\lambda\, \sigma5,\ \_ \bullet$
                 $Q(\sigma5\ /\ \sigma4.\gamma)))))))))$
```

where the upper bounds $\overline{eu}$ contain neither `waitlevel` nor `lockbottom`, and where inv_C is the monitor invariant of the object's class.

Figure 7: Symbolic execution of statements (continued)

```
exec($\sigma$, unshare $e$, $Q$) =
  eval($\sigma$, $e$, ($\lambda\,\sigma 1$, $t$ •
        $\sigma 1 \vdash t \neq \mathit{null}$
      $\wedge$ $(\mathit{holds}(t,\ \sigma 1.h(v_h))\ =\ W)$
      $\wedge$ $\exists\ \mathit{fc}\ @\ t.\mathtt{mu} \mapsto \_\ \#\ t\alpha \in \sigma 1.h$
      $\wedge$ $\sigma 1 \vdash t\alpha \geq 100\ \wedge$
       let $v = \sigma 1.h(v_h) + 1$ in
         $Q(\sigma 1\ /\ \sigma 1.h[v_h{+}{+}]$
               $-\ \mathit{fc} + t.\mathtt{mu} \mapsto \mathit{lockbottom}\ \#\ t\alpha$
               $+\ \mathit{mu}(t,\ v)\ =\ \mathit{lockbottom}$
               $+\ \forall o \neq\ t \cdot \mathit{mu}(\ t,\ v) = \mathit{mu}(\ t,\ v-1)$
               $+\ (\mathit{holds}(t,\ v)\ =\ N)$
               $+\ (\forall o \neq\ t \cdot \mathit{holds}(\ t,\ v) = \mathit{holds}(\ t,\ v-1)))))$

exec($\sigma$, acquire($e$, $m$), $Q$) =
  eval($\sigma$, $e$, ($\lambda\,\sigma 1$, $t$ •
    eval($\sigma 1$, $m$, ($\lambda\,\sigma 2$, $\mathit{tm}$ •
      $\sigma 2 \vdash t \neq \mathit{null} \wedge$
        consume($\sigma 2$, $\mathit{full}$, waitlevel << $e$.mu, ($\lambda\,\sigma 3$, $\_$ •
          let
            $t\alpha$ = if $(\mathit{tm} = W)$ $\mathit{full}$ else $\mathit{eps}$,
            $v = \sigma 3.h(v_h) + 1$,
            $\sigma 4 = \sigma 3\ /\ (\mathit{this},\ t)\ /\ \sigma 3.h[v_h{+}{+}]$
                   $+\ (\mathit{holds}(t,\ v)\ =\ \mathit{tm})$
                   $+\ (\forall o \neq\ t \cdot \mathit{holds}(\ t,\ v) = \mathit{holds}(\ t,\ v-1))$
          in
            produce($\sigma 4$, $\mathit{fresh}$, $t\alpha$, $\mathit{inv}_C$, ($\lambda\,\sigma 5$ •
              $Q(\sigma 5\ /\ \sigma 2.\gamma)))))))))$

exec($\sigma$, release($e$, $m$), $Q$) =
  eval($\sigma$, $e$, ($\lambda\,\sigma 1$, $t$ •
    eval($\sigma 1$, $m$, ($\lambda\,\sigma 2$, $\mathit{tm}$ •
      $\sigma 2 \vdash t \neq \mathit{null}\ \wedge\ (\mathit{holds}(t,\ \sigma 2.h(v_h))\ =\ \mathit{tm}) \wedge$
        let $t\alpha$ = if $(\mathit{tm} = W)$ full else eps in
          consume($\sigma 2\ /\ (\mathit{this},\ t)$, $t\alpha$, $\mathit{inv}_C$, ($\lambda\,\sigma 3$, $\_$ •
            let $v = \sigma 3.h(v_h) + 1$ in
              $Q(\sigma 3\ /\ \sigma 2.\gamma\ /\ \sigma 3.h[v_h{+}{+}]$
                   $+\ (\mathit{holds}(t,\ v)\ =\ N)$
                   $+\ (\forall o \neq\ t \cdot \mathit{holds}(\ t,\ v) = \mathit{holds}(\ t,\ v-1)))))))))$

exec($\sigma$, while ($e$) {$\overline{ss}$}, $Q$) =
  let $\gamma b = \sigma.\gamma + \{(x1,\ \mathit{fresh}),\ \ldots,\ (xn,\ \mathit{fresh})\}$ in
    produce($(\gamma b,\ \emptyset,\ \emptyset,\ \sigma.\pi)$, $\mathit{fresh}$, $\mathit{full}$, $\mathit{inv}_W$ && $\mathit{lkch}_W$, ($\lambda\,\sigma 1$ •
      eval($\sigma 1$, $e$, ($\lambda\,\sigma 2$, $t$ •
        exec($\sigma 2[\sigma 2.h\ /\ g] + t$, $\mathit{body}_W$, ($\lambda\,\sigma 3$ •
          consume($\sigma 3$, $\mathit{full}$, $\mathit{inv}_W$ && $\mathit{lkch}_W$, ($\lambda\,\sigma 4$, $\_$ • $\mathit{true}$)))))))
      $\wedge$
  consume($\sigma$, $\mathit{full}$, $\mathit{inv}_W$ && $\mathit{lkch}_W$, ($\lambda\,\sigma 1$, $\_$ •
    produce($\sigma 1[\sigma.h\ /\ g]\ /\ \gamma b$, $\mathit{fresh}$, $\mathit{full}$, $\mathit{inv}_W$ && $\mathit{lkch}_W$, ($\lambda\,\sigma 2$ •
      eval($\sigma 2$, $e$, ($\lambda\,\sigma 3$, $t$ •
        $Q(\sigma 3[\sigma.g\ /\ g] + \neg t)))))))$
```

where $x1, \ldots, xn$ are all local variables that are assigned to in the loop body but that are declared outside of it, i.e. before the loop and where inv_W, body_W and lkch_W are the loop invariant, loop body and loop lockchange clause, respectively.

Figure 7: Symbolic execution of statements (continued)

3.12 VALIDITY

The correctness of the rules for symbolic execution introduced in the previous subsections, especially the rules for synchronous and asynchronous method invocations, rely on the assumption that methods are correct with respect to their pre- and postconditions. In order to justify this assumption we have to prove that this is actually the case, i.e. that all methods are valid, which in turn requires that functions, invariants and all specifications in general are valid. Ensuring the validity of pure functions includes proving their termination; an orthogonal problem that we have postponed as future work.

In order to be valid a program must be well-formed, e.g. all mentioned variables, fields and classes must exist, all expressions must be correctly typed, etc. We assume that these usual requirements are all met, and focus on the only requirements arising from our verification technique: that assertions are self-framing and that there are no null-dereferences. Producing an assertion fails if the assertion is not self-framed, as already described in Section 3.9, and it is thus sufficient to ensure that assertions are producible in order to ensure that they are self-framing. Moreover, producing an assertion also fails if a possibly null-object is dereferenced, and we can therefore conclude that the whole program is well-formed if all assertions can be produced.

Definition 15. Figure 8: A method is valid if the precondition can be produced in any state where i) `this` is a non-null objects and ii) all method arguments and return values exist, if the method body can then be executed in the resulting pre-state, and finally if the postcondition and the lockchange clause hold in the post-state resulting from the body's execution.

$$
\begin{array}{l}
\textsf{let} \\
\quad \gamma = \{(\textit{this},\ \textit{fresh}),\ (x_1,\ \textit{fresh}),\ \ldots,\ (x_n,\ \textit{fresh}), \\
\qquad\qquad\qquad\qquad (y_1,\ \textit{fresh}),\ \ldots,\ (y_m,\ \textit{fresh})\} \\
\textsf{in} \\
\quad \textsf{produce}((\gamma,\ \emptyset,\ \emptyset,\ \{\textit{this} \neq \textit{null}\}),\ \ \textit{fresh},\ \textit{full},\ pre_{C.m},\ (\lambda\,\sigma 1 \bullet \\
\qquad \textsf{produce}(\sigma 1[\sigma 1.h\ /\ g],\ \ \textit{fresh},\ \textit{full},\ post_{C.m}\ \texttt{\&\&}\ lkch_{C.m},\ (\lambda\ \_ \bullet \textit{true})) \\
\qquad\quad \wedge \\
\qquad \textsf{exec}(\sigma 1[\sigma 1.h\ /\ g],\ body_{C.m},\ (\lambda\,\sigma 2 \bullet \\
\qquad\quad \textsf{consume}(\sigma 2,\ \textit{full},\ post_{C.m}\ \texttt{\&\&}\ lkch_{C.m},\ (\lambda\ \_,\ \_ \bullet \textit{true}))))))
\end{array}
$$

where $x1,\ \ldots,\ xn$ are the method arguments and $y1,\ \ldots,\ yn$ are the return values.

Figure 8: Method validation

The first conjunct of Figure 8 corresponds to ensuring well-formedness of the method specification, whereas the second conjunct ensures that the method is valid with respect to its specification. It is not strictly necessary to produce the postcondition in the first conjunct, because it would be produced at call-site when the method is used, thus well-formedness would still be guaranteed. We produce it nevertheless in order to detect all ill-formed assertions, regardless of their actual use.

Definition 16. Figure 9: A function is valid if the precondition can be produced in any state where i) `this` is a non-null objects and ii) all function arguments exist, and if the method body can be evaluated in the resulting pre-state.

```
let
  $\gamma = \{(\mathit{this},\ \mathit{fresh})(\mathit{result},\ \mathit{fresh}),\ (x1,\ \mathit{fresh}),\ \ldots,\ (xn,\ \mathit{fresh})\}$,
  $\sigma = (\gamma,\ \emptyset,\ \emptyset,\ \{\mathit{this} \neq \mathit{null}\})$
in
  $\mathsf{produce}(\sigma,\ \mathit{fresh},\ \mathit{full},\ pre_{C.p},\ (\lambda\ \sigma 1 \bullet$
    $\mathsf{eval}(\sigma 1,\ body_{C.p},\ (\lambda\ \_,\ \_ \bullet$
      $\mathit{true}))))$
```

Figure 9: Function validation

Definition 17. Figure 10: Predicates and monitor invariants are valid if they are well-formed, i.e. if they can be produced in any state where `this` is a non-null object.

$$\mathsf{produce}((\{(\mathit{this},\ \mathit{fresh})\},\ \emptyset,\ \emptyset,\ \{\mathit{this} \neq \mathit{null}\}),\ \mathit{fresh},\ \mathit{full},\ spec,\ (\lambda\ \_ \bullet \mathit{true}))$$

where $spec$ is either a predicate body $body_V$ or a monitor invariant inv_C.

Figure 10: Predicate and monitor invariant validation

Definition 18. A class is valid if all of its members are valid, i.e. if methods, functions, predicates and the monitor invariant are valid.

Definition 19. A program is valid if all comprised classes are valid.

3.13 SNAPSHOTS AND CHALICE'S UNSOUNDNESS

The vcg-based Chalice verifier is currently unsound in the way it handles predicates. Listing 8 shows an illustrating example which is incorrectly verified by Chalice but not by Syxc. In summary, method `proveFalse()` verifies because neither exhaling nor inhaling (consume and produce in our terminology) havoc field `x` encapsulated by predicate `V`. Unfold would, but no unfolding is done while verifying method `proveFalse()`. Hence, `set(2)` establishes that `get() == 2` which incorrectly remains valid when `set(3)`, which completely consumes access to `V` and thus can and actually will modify the field covered by it, establishes that `get() == 3`. Seen from a different point of view, one could say that the framing of function applications depending on predicates is insufficient because it does only consider the presence of the predicate but not the fields hidden by it.

Listing 8: Unsoundness in the vcg-based Chalice verifier

```
class Unsound {
  var x: int

  predicate V { acc(x) }

  function get(): int
    requires V
  { unfolding V in x }

  method set(x: int)
    requires V
    ensures V && get() == x
  {
    unfold V
    this.x := x
    fold V
  }

  method proveFalse()
    requires V
  {
    call set(2)
    call set(3)

    assert get() == 2 && get() == 3
      /* Correctly fails in Syxc, verifies in Chalice */
  }
}
```

Snapshots overcome this unsoundness by framing predicates and function applications more fine-grained. This is illustrated in Listing 9. Summarised, `set(2)` completely consumes access to `V` and produces a fresh instance of `V`, i.e. a predicate chunk with a fresh snapshot. It also produces a function application term versioned with the same fresh snapshot and relates this term to `2`. Symbolically executing `set(3)` proceeds analogously and thus outdates the previous function application term by invalidating its frame.

Listing 9: Snapshots framing predicates and function applications

```
method proveFalse()
  requires V
{
    $h : t.V[t_1]$
  call set(2)
    $h : t.V[t_2]$
    $\pi : C.get(t_2, t) = t_2, C.get(t_2, t) = 2$
  call set(3)
    $h : t.V[t_3]$
    $\pi : C.get(t_3, t) = t_3, C.get(t_3, t) = 3$

  assert get() == 3 /* Holds */
    $\sigma \vdash C.get(t_3, t) = 3$

  assert get() == 2 /* Fails */
    $\sigma \nvdash C.get(t_3, t) = 2$
}
```

3.14 Sort wrappers

The snapshot of a function application term is the result of the consumption of the function's precondition. Similarly, unfolding a predicate produces the predicate body using the predicate snapshot to determine the values of fields occurring in the body. Snapshot terms are always of sort *Snap*, whereas fields can be of different sort, e.g. *Int*, *Bool* or *Ref*. This can lead to sort incompatibilities, as illustrated in Listing 10.

Listing 10: A snippet illustrating the need for sort wrappers

```
class C {
  var a: bool

  function neg(): bool
    requires acc(a)
  { !a }

  method appNeg() returns (b: bool)
    requires acc(a)
  {
      // $\gamma : b \mapsto tb$
      // $h : t.a \mapsto ta$    where $ta$ is of sort $Bool$
    b := neg()
      // $\gamma : b \mapsto C.neg(\neg ta, t)$    error, snapshot must be of sort $Snap$
      // $\pi : C.neg(ta, t) = \neg ta$
  }

  predicate V { acc(a) }

  method unfoldV()
    requires V
  {
      // $h : t.V[ta]$    where $ta$ is of sort $Snap$
    unfold V
      // $h : t.a \mapsto ta$    error, $ta$ must be of sort $Bool$
  }
```

In order to avoid this problem we include in our logic's signature two sort wrapper functions toSnap_S and $\mathit{fromSnap}_S$ for each sort S:

$$\begin{aligned}&\mathit{toSnap}_S \colon S \to \mathit{Snap}\\&\mathit{fromSnap}_S \colon \mathit{Snap} \to S\end{aligned}$$

When a field access assertion `acc(x.f)` is produced with a snapshot $t_v : \mathit{Snap}$ the resulting field chunk points to t_v as the field's value, but appropriately wrapped to match the field value's sort S, i.e. $t_x.f \mapsto \mathit{fromSnap}_S(t_v) \;\#\; t_\alpha$. The inverse wrapper function is applied when a field access assertion is consumed. If the field's value has been t_v then the term yielded by the consumption will be $\mathit{toSnap}_S(t_v)$. Accordingly, our theory $\mathcal{T}$ contains axioms defining the corresponding to- and from-wrappers as mutually inverse:

$$\begin{aligned}&\forall\, t\colon S \bullet \mathit{toSnap}_S(\mathit{fromSnap}_S(t)) = t\\&\forall\, t\colon S \bullet \mathit{fromSnap}_S(\mathit{toSnap}_S(t)) = t\end{aligned}$$

For the sake of brevity we have omitted the application of these wrapper functions from our symbolic execution rules for the production and consumption of field access permissions.

3.15 SHARED OBJECTS

Careful readers might ask themselves why mu-fields are encoded as regular field chunks and additionally by the mu-function as part of the path conditions, and why the holds-function is versioned by a counter and not by a snapshot, as are all other functions. These are valid questions, especially since each function update results in a quantified term being added to the path conditions, thwarting the initial goal of symbolic execution to reduce the number of quantified formulae that the theorem prover has to deal with. However, we have good reasons for our taken design decisions.

Mu-fields are regular fields in the sense that access to them has to be required by e.g. `acc(c.mu)` assertions, which is why we have corresponding mu-field chunks. At the same time, waitlevel-comparisons such as `waitlevel << c.mu` correspond to $\forall\, r \bullet holds(r) \Rightarrow mu(r) < mu(c)$ (simplified) and are therefore part of the path conditions. The quantification ranges over possibly infinitely many (unknown) object references, but our heap is always finite and mentions only those objects that we explicitly know about. Hence, we need to encode mu-fields in a way that suits both purposes.

Versioning applications of the holds-function by snapshots does not seem to be possible since snapshots are created by consuming the precondition of a function, but it is not apparent how a suitable precondition for the holds-function would look like and if it can be expressed in Chalice. Since the holds-status of an object may change during program execution we nevertheless need to be able to update the holds-function, and revision numbers seem to be a good way of achieving this.

4 Syxc

This section gives a brief overview of Syxc, our implementation of the symbolic execution algorithm presented in Section 3. Syxc has not yet been released, but we plan to do so once the thesis has been submitted.

4.1 Overview

Syxc, our implementation of the symbolic execution algorithm presented in Section 3, is written in Scala (version 2.8.0). It uses the existing Chalice parser to typecheck, parse and resolve the program to verify. The generated abstract syntax tree (AST) is converted into a very similar, but nevertheless slightly different AST representation. This gives us the possibility to report yet unsupported features such as channels, and to rewrite and simplify certain AST nodes. In the long term, it makes Syxc independent from Chalice in the sense that we could implement converters from other languages to our AST and thereby verify these languages, too. That way Syxc could be used as a general verification engine, as e.g. Boogie is used. In order to facilitate future extensions, Syxc has been designed with modularity in mind. Most interfaces are type-parameterised (generics), in particular state and verifier classes.

Syxc depends on a two-level decision unit hierarchy to discharge arising proof obligations. The first level, a so-called *decider*, is queried by Syxc directly and provides several specific functions that decide e.g. if permissions grant write access or if a lock can be acquired. In order to make such decisions, the decider may or may not make us of a general theorem prover. In our current implementation we use Z3, but since our term hierarchy is independent of the underlying prover we could replace Z3 by any other suitable theorem prover.

4.2 Components

Syxc roughly consists of four major parts: the state, the actual verifier, the decision units and the algebraic datatype hierarchies AST nodes and terms, on which the verifier operates. A sketch of the general structure is shown in Figure 11. AST nodes and terms comprise an abstract base class – `ASTNode` and `Term`, respectively – and many concrete nodes and terms, e.g. `Class`, `Statement`, `Expression` and, respectively, `FApp`, `Unit`, `UpdateHolds`. For each abstract state, verifier or decision unit class there currently exists one concrete implementation, e.g. the `DefaultProducer` implementing the `Producer` interface, or the `SetBackedPathConditions` implementing the `PathConditions` interface.

All state classes take at least one type parameter, the so-called *MyType* ([Bru93], illustrated in Listing 26 in the appendix), which enables state classes to reference objects of their own concrete type in a type-safe manner. The state interface is

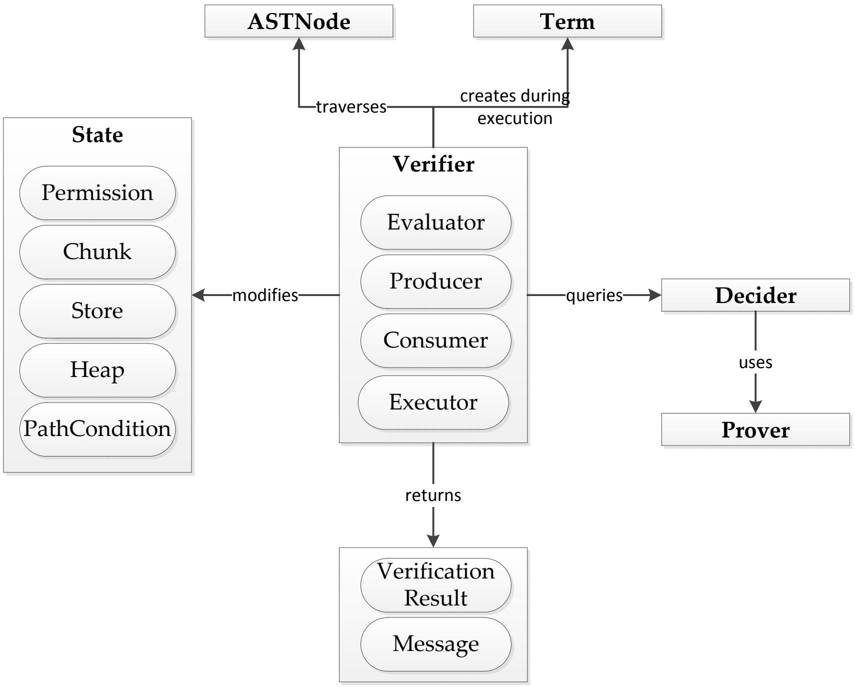

Figure 11: Components of Syxc

composed of three smaller interfaces (`HasStore`, `HasHeaps`, `HasPathConditions`), each declaring the operations available on their particular component. See Listing 27 in the appendix for the major state class interfaces. Operations on collection-like structures are uniformly named \+ and \– to indicate adding and removing of elements, respectively, and \ to indicate substitution. Combined with Unicode identifiers this results in a strong visual correspondence between the rules as typeset in Section 3 and as implemented in Syxc. All current state class implementations are immutable, which facilitates reasoning by developers and thus maintenance, and also contributes to the visual correspondence between typeset and implemented rules. See Listing 11 for an example of how the implemented rules look like.

Listing 11: An example of an implemented rule

```
// ... Setting up local variables ...

/* Verify loop body (including well-formedness check) */
val σW = ∅ \ (γ = bodyγ, π = σ.π)
(produce(σW, fresh, Full, inv, specsErr, σ1 =>
  eval(σ1, guard, m, (σ1a, tGuard) =>
    exec(σ1a \ (g = σ1a.h) \+ tGuard, body, m, σ2 =>
      consume(σ2, Full, inv, InvNotPreserved, (σ3, _) =>
        Success))))
  &&
/* Verify call-site */
consume(σ, Full, inv, InvNotEstablished, (σ1, _) => {
  val σ2 = σ1 \ (g = σ.h, γ = bodyγ)
  produce(σ2, fresh, Full, inv, m, σ3 =>
    eval(σ3, guard, m, (σ4, tGuard) =>
      Q(σ4 \ (g = σ.g) \+ ¬(tGuard))))}))
```

The verifier itself consists of the interfaces `ProgramVerifier` and `MemberVerifier`, where the former is intended to rely on the latter in order to verify complete programs. A direct consequence of this design is that members, especially methods, can be verified in parallel by forking several member verifiers. Member verifiers are intended to utilise an `Evaluator`, a `Producer`, a `Consumer` and an `Executor` to symbolically execute the program. These interfaces are again heavily type-parameterised, as can be seen in Listing 28 in the appendix. The current implementations, called `DefaultEvaluator` etc., are almost completely independent of concrete state classes, including permissions. The only exception is the `DefaultEvaluator` that depends on the current concrete implementation of permissions, since it evaluates Chalice permission AST nodes into Syxc permission terms. Since the permission model of Chalice is currently being revised[1], it is important to reference the current permission term implementation as seldom as possible. However, there are situations where referencing it cannot be avoided, e.g. in the `DefaultEvaluator` or the `DefaultDecider`.

Evaluator, producer, consumer and executor have been implemented as Scala traits with self-type annotations declaring mutual dependencies. An alternative would have been regular dependency injection[2], but:

1. Constructor dependency injection would not have been possible, because e.g. the `DefaultEvaluator` depends on a `Consumer` and the `DefaultConsumer` depends on an `Evaluator`. Thus, neither object could have been instantiated.

2. Setter injection would have been possible, but results a) in temporarily incomplete, i.e. only partly initialised objects, which is error-prone, b) obfuscates object initialisation by lots of additional setter-invocations, and c) also makes the implemented rules less readable, for example by turning invocations of `eval(...)` into `evaluator.eval(...)`.

The decider is queried by the aforementioned verifier classes in order to prove or refute that an assertion follows from the current state. Since this in general requires support from a theorem prover, the decider usually proceeds by converting the state, especially the gathered path conditions, into a format understood by the prover. If appropriate, the decider may delegate tasks to other decision procedures which are possibly better suited for specific tasks, e.g. numeric constraint solvers, or to not delegate tasks at all. Our current decider, for example, distinguishes between permissions that contain integer literals only, e.g. `acc(f, 50)`, and those that include general expressions, e.g. `acc(f, n/2)`. When e.g. comparing two permissions, it invokes Z3 only in the latter case.

4.3 Additional features

In addition to the Chalice features covered in Figure 1 and by the rules presented in Section 3.8 ff., Syxc also supports sequences and universally and existentially quantified expressions. See Listing 12 for an illustrating example.

[1] http://www.pm.inf.ethz.ch/education/theses/student_docs/Stefan_Heule/Stefan_Heule_RCS_Description

[2] http://martinfowler.com/articles/injection.html

Listing 12: Sequences and quantified expressions

```
class Test {
  method neg(xs: seq<int>) returns (ys: seq<int>)
    requires forall i in [0..|xs|] :: xs[i] > 0
    ensures forall i in [0..|ys|] :: ys[i] < 0
  {
    var i: int := 0
    var x: int
    ys := []

    while (i < |xs|)
      invariant 0 <= i
      invariant forall j in [0..|ys|] :: ys[j] < 0
    {
      x := -xs[i]
      ys := ys ++ [x]
      i := i + 1
    }
  }
}
```

4.4 Test suite

In order to facilitate maintenance and future development, and to generally increase confidence in the developed verifier, Syxc includes an extensive test suite covering all supported Chalice constructs. In contrast to the existing Chalice test suite, which is considerably smaller but instead contains several intricate test cases, Syxc's suite is intended to quickly uncover deviations from the expected verification results. Syxc also includes an analyser that compares the test output with annotations made to the test cases and reports errors if the output does not match the annotated behaviour. See Listing 13 for an illustrating example. So far, the analyser reports thrown exceptions and it supports the following annotations:

- `@Error` n, where n is an error code. If the expected error occurs, nothing is reported. Otherwise, the analyser distinguishes between the complete absence of an error and mismatching ones.
- `@Fails`, which denotes a line/an assertion that is correct but does not (yet) verify. The analyser will report an unexpected behaviour should the underlying incompleteness be resolved – by purpose or by accidentally introducing an unsoundness –, thereby facilitating maintenance of the test suite and future development in general.
- Similarly, `@Holds` denotes a line/an assertion that should fail but currently doesn't. This annotation is intended to mark results of known unsoundnesses such that they can be spotted easily. Again, the analyser will issue a warning once the unsoundness has been fixed.

Errors occurring in lines that have not been annotated will be reported by the analyser as unexpected errors.

Listing 13: A test case and the output generated by the test case analyser

```
/* ---------- Two failing test cases ---------- */
method fails1()
  requires rd(this.mu)
  ensures acc(this.mu) /* @Error 330 */
{}

method fails2()
  requires rd(this.mu)
{ assert rd(this.mu, 2) } // Will fail

/* ---------- Analyser output ---------- */
/*
===== Found unexpected errors in 1 file(s)! =====

testsuite\example_test_case.chalice
  Error 430: 4.4: Assertion might not hold at 4.11.
             Insufficient permissions to access this.mu.
*/
```

At the time of writing, Syxc includes more than 100 test cases with round about 750 methods, 50 functions, 40 predicates, and 300 `@Error`, no `@Holds` and 25 `@Fails` annotations. Most of the latter are due to sequences and a lot of those also fail in the vcg-based verifier. Runtimes of both verifiers are presented in Table 3.

5 RESULTS

5.1 CHALICE TEST SUITE

In order to assess Syxc with respect to completeness and overall performance, Syxc has been tested against the Chalice test suite found in the `examples` directory. This set contains 29 test cases, of which 14 have been ignored for reasons itemised in Table 1. Syxc has been tested against the remaining 15 test cases, with results itemised in Table 2. See Section 5.3.1 for a description of what is being referred to as the ”chunk lookup & fapp-term” incompleteness.

Table 1: Unconsidered test cases from the existing Chalice test suite

Files	Reason
prog0	does not typecheck
RockBand-automagic, cell-defaults	intended to test automagic-features of Chalice that are not supported by Syxc, e.g. `-autoFold`
AssociationList	reordering of locks is not yet supported by Syxc
counter, ForkJoin	eval-expressions are not yet supported by Syxc
prog3	eval-expressions, `rd(x,*)` and `waitlevel == old(waitlevel)` are not yet supported by Syxc
quantifiers	access permissions ranging over arrays, e.g. `rd(arr[*].x)`, are not yet supported by Syxc
CopyLessMessagePassing[2], ImplicitLocals, ProdConsChannel, Sieve	channels are not yet supported by Syxc

Comparing the number of errors reported by Chalice with the number reported by Syxc is not really meaningful due to the way the two verifiers proceed in the presence of errors. When an assertion fails, Chalice continues verifying the method or function assuming that the assertion holds. Syxc, in contrast, immediately aborts the current verification and continues with the next method or function. Another difference in error handling can be perceived when a postcondition is ill-formed, e.g. not self-framing. The vcg-based verifier will report this when the assertion's well-formedness is checked. When it verifies a corresponding call-site, however, it apparently ignores the assertion and therefore e.g. does not havoc certain fields which may result in a contradiction. Syxc, on the other hand, will report the assertion as ill-formed and will again report an error when verifying the call-site.

[2] CopyLessMessagePassing, CopyLessMessagePassing-with-ack and CopyLessMessagePassing-with-ack2

Table 2: Testing Syxc against the existing Chalice test suite

File	Results
RockBand	verifies when desugaring `acc(x.*)` and adding explicit folds and unfolds
PetersonsAlgorithm, OwickiGries, linkedlist, prog1, prog4, swap, cell	verify
LoopLockChange	mostly verifies; additional non-null clauses necessary; `Test5` fails due to the incompleteness described in Section 5.3.2
prog2	mostly verifies; removed assigned-expressions test case; one expected error is missing due to unsupported `ghost const` field modifiers; one additional error due to different handling of ill-formed assertions
HandOverHand	at least two unexpected lockchange-errors for reasons yet to be determined; unsupported assigned-expressions can be removed (still verifies in Chalice)
iterator, iterator2	fail due to known "chunk lookup & fapp-term" incompleteness; additional non-null clauses necessary
dining-philosophers	fails due to known "chunk lookup & fapp-term" incompleteness; desugaring of `acc(x.*)` necessary
producer-consumer	fails due to known "chunk lookup & fapp-term" incompleteness

During the development of Syxc several bugs or shortcomings have been discovered in the vcg-based Chalice verifier. Some of these merely regard unsupported features, e.g. directly forking into a token field instead of a token variable. Others uncovered bugs in the implementation and yet others revealed surprising aspects of the semantics of certain constructs, e.g. old-expressions, and might lead to a reevaluation thereof. Due to time constraints, most bugs have not yet been reported to the Chalice team, but we kept track of them and plan to file reports once this thesis has been submitted.

5.2 RUNTIME PERFORMANCE

We benchmarked Syxc and Chalice in order to compare their runtime performance, with results presented in Table 3. As one can see, Syxc in general outperforms Chalice and is often twice as fast, with the striking exception of Peterson's algorithm, where it performs particularly worse. The reasons for this exception are yet to be determined. All runtimes presented in this report are the average of three subsequent verification runs on an Intel Core2 Quad CPU Q9550, 2.83GHz, 4GB RAM running Windows 7 x64 and Chalice Rev64765, Boogie Rev63071, Z3 2.16, all three with default settings.

We also tested Syxc and Chalice on two additional, rather artificial test cases. The first test case corresponds to the test case in [SJP09b], page 14 bottom, and it subse-

Table 3: Comparing runtime performance

Test(s)	Syxc	Chalice	Comment
Syxc test suite (100 files)	311s	440s	Test category "fast"; Syxc verifies all cases as expected, but this has not been checked for Chalice, which could influence runtime performance
RockBand	1.04s	3.04s	
PetersonsAlgorithm	137.80s	6.59s	
OwickiGries	1.22s	2.66s	
linkedlist	2.02s	2.61s	
prog1	1.04s	2.60s	
prog4	1.02s	2.51s	
swap	0.88s	2.24s	
cell	1.68s	3.32s	
LoopLockChange	1.38s	2.94s	
prog2	1.02s	2.54s	

quently instantiates 20 cells with 20 different values and finally asserts that all of them have retained their initial value. Syxc needed less than one second to verify this test case, whereas the vcg-based verifier needed 4.49s. The second test case essentially is a huge nested if-then-else statement resulting in 10^4 disjoint execution paths. Our initial assumption has been that Syxc would perform significantly worse than Chalice when challenged with such an amount of execution branches, and runtimes of 617s (Syxc) vs. 330.44s (Chalice) apparently prove us right. However, the worse runtime of Syxc might partly result from the inefficient interaction with Z3, as described in Section 6.2.

5.3 COMPLETENESS

While not being the only criterion for the assessment of an automatic verification technique, completeness nevertheless is one of the most important ones. In this subsection we will therefore present cases where Syxc succeeds and the vcg-based verifier fails, or vice versa.

5.3.1 *Function applications and heap lookups*

A crucial but presumably not unsolvable incompleteness of the current state of our symbolic execution algorithm arises from the way heap chunks are looked up. For example, looking up a field chunk for a receiver t and a field f in a heap h succeeds only if there exists a chunk $t.f \mapsto \_ \# \_$ in h. That is, the lookup will fail if $t = t'$ is part of the path conditions and $t'.f \mapsto \_ \# \_$ is a chunk in the heap. As a consequence, the verification of methods relying on getter-functions often fails, as illustrated in Listing 14. In the example, `assert acc(get().x)` fails, although Syxc can prove all necessary intermediate steps, i.e. `acc(d.x) && d == get() == c`. Evaluating the function application `get()` yields the term $\mathit{Fapp}(t_d, t, \mathit{get})$ and adds the equality $\mathit{Fapp}(t_d, t, \mathit{get}) = t_d$ to the path conditions. However, since these are not considered when asserting that the current thread has access to $\mathit{Fapp}(t_d, t, \mathit{get}).x$, the latter will fail.

Listing 14: Illustrating Syxc' heap lookup incompleteness I

```
class Test {
  var c: Cell

  function get(): Cell
    requires rd(c)
  { c }

  method test(a: int)
    requires acc(c) && c != null && acc(c.x)
  {
    c.x := a
    // $\gamma : this \mapsto t, a \mapsto t_a$
    // $h : t.c \mapsto t_c, t_c.x \mapsto t_a$

    assert get() == c /* Holds */
    assert get().x == a
      /* Fails in Syxc, holds in Chalice */
    // eval(get()) $= \mathit{Fapp}(t_a, t, \mathit{get})$
    // and $\pi : \mathit{Fapp}(t_a, t, \mathit{get}) = t_a$
    // but $\mathit{Fapp}(t_a, t, \mathit{get}).x \mapsto t_a \notin h$
  }
}
```

A possible solution to this incompleteness could be to include the prover in the decision whether a certain heap chunk exists. It is not sufficient, though, to just know *that* a necessary heap chunk exists, but rather *which* chunk that is. This is due to the necessity that we must be able to remove that particular chunk from the heap when consuming corresponding assertions. Using Z3's model-finding capabilities in a manual test encoding of such a situation took a promising course, but requires references to be elements of a finite set in order for Z3 to find the correct chunk. The disadvantage of this approach is, of course, that it increases the number of Z3 invocations and, due to model-finding, also the complexity of the operations to perform. Another approach would be invoke Z3 separately for each heap chunk in order to assert equality between the current chunk at hand and the chunk that is to be looked up.
An alternative might be to perform term substitution on the state whenever an equality such as $t = t'$ is added to the path conditions. In the situation above, each occurrence of $\mathit{Fapp}(t_d, t, \mathit{get})$ could be replaced by t_d (or the other way round), including the term that is returned by the evaluation of `get()`. This approach is probably more efficient in terms of performance and does not constrain the choice of a prover to those having sufficient model-finding capabilities.

Another incompleteness arising from the separation of heap and path conditions is illustrated in Listing 15. Here, $\mathit{Fapp}(t_c, t, \mathit{get}, 0).x \mapsto t_x \;\#\; \_ \in h$ and $t_a = 0 \in \pi$, but the example nevertheless fails in Syxc, because path conditions are not considered when looking up heap chunks.

Listing 15: Illustrating Syxc' heap lookup incompleteness II

```
function get(a: int): Cell
  requires rd(c)
{ c }

method fails(a: int)
  requires rd(c) && get(0) != null && acc(get(0).x)
  requires a == 0
{
  // $\gamma : \mathit{this} \mapsto t, a \mapsto t_a$
  // $h : \mathit{FApp}(t_c, t, 0).x \mapsto t_x$
  assert acc(get(0).x) /* Holds */
  assert acc(get(a).x)
    /* Fails in Syxc, holds in Chalice */
}
```

5.3.2 *While loops*

There are three tasks to perform when verifying a while loop: 1) ensure that the specification, i.e. the invariant, is well-formed, 2) verify the loop body with respect to the invariant and 3) verify the call-site, i.e. ensure that the invariant holds before the loop is entered. Currently, the first task might fail if the loop invariant branches due to if-then-else-expressions or implications and if the guards/antecedents of those expressions depend on the loop guard.

Such a situation is illustrated by method `fails` in Listing 16, and the reason why Syxc fails to verify the example is the following: The loop guard has to be evaluated in order to assume that it holds before the loop body is executed. Since the loop guard dereferences field `x` the invariant has to be produced first to ensure that `x` is accessible. Producing the invariant, however, branches on the implication since the guard has not yet been assumed. The execution of the branch that assumes `a` correctly verifies, but the execution of the branch that assumes `!a` fails, because `acc(x)` has not been produced and the evaluation of the guard therefore fails.

Method `inconsistent` in Listing 16 illustrates that the aforementioned problem of mutually dependent loop guard and invariant can even lead to inconsistent path conditions. This, however, is a spurious unsoundness in the sense that the branch that verifies due to inconsistent path conditions is actually never executed.

A possible solution to this problem might be to partition the guard's conjuncts into heap-dependent and -independent ones, and to assume the latter before producing the invariant and assuming the heap-dependent conjuncts.

Listing 16: Illustrating Syxc's while loop incompleteness

```
class Test {
  var x: int

  method fails()
    requires acc(x)
  {
    var a: bool := true
    /* Fails in Syxc, holds in Chalice */
    while (a && x > 0)
      invariant a ==> acc(x)
    { a := false }
  }

  method inconsistent()
    requires acc(x)
  {
    var a: bool := true

    while (a)
      invariant a ==> true
    {
      assert !a ==> false
        /* Holds in Syxc and in Chalice,
         * but for different reasons
         */
    }
  }
}
```

5.3.3 *Recursive predicates*

Listing 17 illustrates an incompleteness of the vcg-based verifier that does not exist in Syxc and that manifests itself when dealing with recursive predicates. We assume that the incompleteness arises from the current handling of predicates, but the exact reason for it is yet to be determined. It is likely that the incompleteness will be overcome soon since the predicate handling is currently being reconsidered.

Listing 17: Illustrating Chalice's recursive predicate incompleteness

```
class Node {
  var v: int
  var next: Node

  predicate V {
    acc(v) && acc(next) && (next != null ==> next.V)
  }

  function length(): int
    requires rd(V)
  {
    1 + unfolding rd(V)
        in next == null ? 0 : next.length()
  }

  method test()
    requires V
    ensures V
  {
    unfold V  /* Comment unfold ... */
    fold V    /* ... and fold and the assertion holds */
    assert length() == old(length())
      /* Holds in Syxc, fails in Chalice */
  }
}
```

5.3.4 *Nested predicates*

In addition to the previous incompleteness, there exists another predicate-related incompleteness in the vcg-based verifier. In the example presented in Listing 18, the second fold-statement fails due to insufficient permissions to access `Test.Z`. This is not the case in Syxc, which correctly verifies the example. As with the previous incompleteness, the reason for this one is also yet unknown.

Listing 18: Illustrating Chalice's nested predicates incompleteness

```
class Test {
  var z: int
  predicate Z { acc(z) }
  predicate ZZ { Z }

  method fails()
    requires ZZ
  {
    unfold acc(ZZ, 40)
    unfold acc(Z, 20)
    fold acc(Z, 10)
    fold acc(ZZ, 30)
      /* Holds in Syxc, fails in Chalice */
  }
}
```

5.3.5 *Distributing access over multiple predicates*

When the current thread gains a predicate chunk, Syxc currently does not inspect the predicate body in order to relate it with other predicates the current thread already has access to, namely to those that hide access to the same fields. Listing 19 illustrates that this gives rise to an incompleteness. The key issue here is that the predicate chunks $t.\mathtt{X1}[t_v]\ \#\ \_$ and $t.\mathtt{X2}[t_w]\ \#\ \_$ each include their own snapshot determine the value of `x`. Thus, `y1` is set to t_v and `y2` to t_w, and it is unknown whether they are equivalent or not. Setting `y1` and `y2` according to the commented lines, however, adds $t_v = t_w$ to the path conditions and thereby makes the example verify. We assume that such additional unfolding-expressions can be inferred automatically, but the conjecture has not been tested yet.

Listing 19: Illustrating Syxc' distributed access incompleteness

```
class Test {
  var x: int

  predicate X1 { acc(x, 40) }
  predicate X2 { acc(x, 60) }

  method fails()
    requires X1 && X2
    ensures X1 && X2
  {
    var y1: int := unfolding X1 in x
    // var y1: int := unfolding X1 in unfolding X2 in x

  var y2: int := unfolding X2 in x
  // var y2: int := unfolding X2 in unfolding X1 in x

  assert y1 == y2
   /* Fails in Syxc, holds in Chalice */
 }
}
```

5.3.6 *Recursive functions*

The linked list snippet presented in Listing 20 verifies in Syxc, but does not verify in Chalice. According to Rustan Leino[3], one of the creators of Chalice, this is due to so-called *limited functions*:

> ”In general, limited functions are used to curb the SMT-solver’s appetite for instantiate quantifiers. [...] The general idea is this: Suppose a user declares a function `F`, defined recursively as follows:
>
> ```
> function p(x): T {
>  if b then t else F(y)
> }
> ```
>
> where I’m using `b`, `t`, and `y` to stand for expressions that may involve `x`. The straightforward axiom for this function is:
>
> ```
> (forall x ::F(x)== if b then t else F(y))
> ```
>
> but this may give rise to matching loops. So, instead, two functions are introduced for the SMT-solver: `F` and `F#limited`, alongside the following axioms:
>
> ```
> (forall x ::F(x)== F#limited(x))
> (forall x ::F(x)== if b then t else F#limited(y))
> ```
>
> Logically, these say the same thing about `F` as the straightforward axiom above. The difference lies in which triggers are used with the quantifiers. They are:
>
> ```
> (forall x :: { F(x) }   F(x) == F#limited(x))
> (forall x :: { F(x) }   F(x) == if b then t else F#limited(y))
> ```
>
> So, if occurrences of `F` in the user’s program are translated into `F`, only those functions will cause instantiations of the axioms. This means that the SMT-solver will not know the function’s definition for recursive calls.
>
> [...]
>
> But maybe limited function are not actually needed in Chalice. That is, perhaps the perceived need for them was a premature conclusion. Maybe the permissions in Chalice have some effect similar to that achieved by limited functions in the first place.”

[3] Source: personal e-mail communication, 14th February 2011

Limited functions have been introduced to Chalice in August 2010, and older versions of Chalice correctly verify the linked list snippet.

Listing 20: Illustrating Chalice’s function application incompleteness

```
class Node {
  var v: int
  var next: Node

  predicate V {
    acc(v) && acc(next) && (next != null ==> next.V)
  }

  function length(): int
    requires rd(V)
  {
    1 + unfolding rd(V)
        in next == null ? 0 : next.length()
  }

  function at(i: int): int
    requires rd(V)
    requires i >= 0
    requires i < length()
  {
    unfolding rd(V) in
      i == 0
        ? v
        : next.at(i - 1)
          /* Holds in Syxc, fails in Chalice */
  }
}
```

5.3.7 *Waitlevel*

A minor incompleteness of the vcg-based verifier reveals itself in the snippet presented in Listing 21. The incompleteness can be overcome by adding the axiom

$$\forall\, n\colon Mu \bullet n \neq \mathit{lockbottom} \Rightarrow \mathit{lockbottom} < n$$

to the Boogie preamble generated by the vcg-based Chalice verifier.

Listing 21: Illustrating Chalice’s waitlevel incompleteness

```
method test(c: Cell)
  requires c != null && rd(c.mu) && lockbottom != c.mu
  requires waitlevel == lockbottom
{
  assert waitlevel << c.mu
    /* Holds in Syxc, fails in Chalice */
}
```

Listing 22 presents an incompleteness of Syxc, which currently does not handle expressions of the form `c << waitlevel` correctly. Such an expression corresponds to an existential quantification over currently held objects, but this has not been implemented yet.

Listing 22: Illustrating Syxc's waitlevel incompleteness

```
class Test {
  method test() {
    var t1: Test := new Test
    var t2: Test := new Test

    share t1 above waitlevel
    share t2 above t1
    acquire t2

    assert t1.mu << waitlevel
      /* Fails in Syxc, holds in Chalice */
    unshare t2
  }
}
```

5.3.8 *Lockchange clauses*

The example presented in Listing 23 verifies neither in Syxc nor in Chalice, with the common reason that the lockchange clause might not contain all changed locks. The only changed lock, however, corresponds to a local variable and thus is not visible by method clients. The problem can be solved by unsharing the local object and thus is a rather minor one.

Listing 23: Illustrating lockchange incompleteness

```
method fails() { /* Fails in Syxc and Chalice */
  var c: Cell := new Cell
  share c
  acquire c
  // unshare t /* Solves the problem */
}
```

5.3.9 *Negated holds-expressions*

As stated in Section 2.2, Syxc currently interprets `!holds(c)` and `!rd holds(c)` as `holds(c, N)`. While this simplifies the symbolic execution rules concerned with holds-expressions, it also gives rise to an incompleteness illustrated in Listing 24. The example does not verify, because the third conjunct of the precondition overwrites the second one, i.e. it updates the holds-function.

Listing 24: Illustrating negated holds-expressions incompleteness

```
method fails(c: Cell)
  requires c != null && rd holds(c) && !holds(c)
{
  // assert false /* Fails, i.e. no unsoundness */
  assert !holds(c)
  assert rd holds(c)
    /* Fails in Syxc, holds in Chalice */
}
```

A possible solution would be to introduce a second holds-function, e.g. *rdholds*, so that they can be updated independently, which actually is exactly what the vcg-based verifier does. However, one could also reconsider lock modes in general and replace them by lock permissions. That is, instead of only allowing read and write locks, one would acquire a fraction $0 \leq p \leq 1$ of the lock. That way, the permission model could be reused for locks.

5.3.10 *Sequences*

Sequences are not really a strong point of neither Syxc nor the vcg-based verifier. That said, Listing 25 presents three small cases: one that verifies in Syxc but not in Chalice, and two that fail in both verifiers. See Section 6.2 for additional limitations of Syxc with respect to the verification of sequence operations.

Listing 25: Illustrating sequence incompletenesses

```
class Test {
  var xs: seq<int>

  method concat(a: int)
    requires acc(xs)
    ensures acc(xs)
    ensures |xs| == old(|xs|) + 1
      /* Holds in Syxc, fails in Chalice */
  { xs := xs ++ [a] }

  method length()
    requires acc(xs)
    requires forall x in [3,4,5,6,7,8,9] :: x in xs
  { assert |xs| >= 6 /* Fails in both */ }

  method at(k: int)
    requires k > 0
  {
    assert exists i in [0..k+1] :: [0..k+1][i] == k
      /* Fails in both */
  }
}
```

Chalice includes the possibility to have access assertions range over sequences, denoted by `acc(`e`[*].`f`)`, where e is an expression of type `seq<`C`>`, and where f is a field of class C. This is not yet supported by Syxc, and it poses an interesting problem since it is not obvious how to encode such assertions while preserving the separation between heaps and path conditions.

5.3.11 *Access permissions*

Asserting that the current thread has at least zero permissions to access a field, e.g. `acc(c.x, 0)`, currently fails in Syxc if a corresponding heap chunk does not exist. A possible solution would be to check if the fractional permission is equivalent to zero first, and to only look for a corresponding chunk if that is not the case.

5.4 KNOWN ISSUES

At the time of writing, the following issues with Syxc are known:

- Boolean sequences are not yet supported and

  ```
  var xs: seq<bool> := [true]
  ```

  hence will fail with an exception reporting unexpected prover output. This is due to an insufficient Z3-encoding of sequence elements, which are assumed to be of type `Int`. See Section 3.14 for a possible solution based on sort wrapper functions.

- `waitlevel` inside old-expressions is not yet supported and e.g.

  ```
  ensures waitlevel == old(waitlevel)
  ```

  will fail with an exception reporting a match error.

- Type-quantifications, e.g.

  ```
  requires forall i: int :: 5 <= i && i <= 10 ==> f(i)
  ```

  are not yet supported and will result in a corresponding error message. Such quantifications can sometimes be rewritten in a form supported by Syxc, here for example as

  ```
  requires forall i in [5..11] :: f(i)
  ```

  However, this is in general only possible if the quantification ranges over a finite set.

- Old-expressions in monitor invariants are not yet supported. Their semantics are currently unspecified, thus using them might result in an error being thrown, but it might also have other, arbitrary effects.

6 CONCLUSION

We successfully extended the work of Smans, Jacobs and Piessens [SJP10], yielding a symbolic execution algorithm for the Chalice language that supports fractional permissions and fork-join concurrency with shared mutable data structures, monitor invariants and deadlock-detection. The algorithm has been formalised and implemented in a research tool named Syxc, which already performs quite respectably when challenged with the Chalice test suite, and in most cases outperforms the vcg-based verifier. The symbolic execution rules have been implemented in a way such that they closely resemble their formal counterparts. Syxc has been designed with extensibility and maintainability in mind and includes an extensive test suite covering the supported language constructs.

Reviewing our goals from Section 1.1, we can conclude that we definitely gained first-hand experience with symbolic execution, that we were able to identify reasons for incompletenesses of our technique, and that we even gained further understanding of Chalice and the vcg-based verifier. However, we cannot yet clearly decide if the symbolic-execution-based approach or the vcg-based approach is superior and thus the technique of choice for the development of Scala verifier. The answer probably depends on the ability to resolve the known incompletenesses and to determine the reasons for the significant difference in runtime between Syxc and Chalice when challenged with the implementation of Peterson's algorithm (Table 3).

6.1 RELATED WORK

Our work is, as a matter of course, closely related to the vcg-based Chalice verifier developed by Leino and Müller [LM09], Leino [Lei10], Leino, Müller and Smans [LMS10], and to VeriCool 3[1] and SpecCheck[2] developed by Smans, Jacobs and Piessens [SJP09b, SJP10]. We thoroughly compared our work to Chalice in Section 5, and we described our extension relative to VeriCool in the previous paragraphs. The incompleteness described in Section 5.3.1 does not seem to be a problem for the latter.

Boyland [Boy03] introduced fractional permissions, which Chalice uses in combination with implicit dynamic frames introduced by Smans, Jacobs and Piessens [SJP09a] to frame methods, to prohibit data races and to allow shared read access to data structures.

Several tools using symbolic execution to verify programs specified in separation logic have been developed in recent years: Smallfoot[3], developed by Berdine, Calcagno and O'Hearn [BCO05], which pioneered this idea, and jStar[4] developed by

[1] http://people.cs.kuleuven.be/~jan.smans/vericool3/
[2] http://people.cs.kuleuven.be/~jan.smans/speccheck/
[3] http://www.dcs.qmul.ac.uk/research/logic/theory/projects/smallfoot/
[4] http://www.jstarverifier.org/

Distefano and Parkinson [DP08] and VeriFast[5] developed by Jacobs and Piessens [JP08], targeting Java and C.

The relationship between separation logic and implicit dynamic frames is currently being investigated by Parkinson and Summers [PS11] and might allow specifications based on dynamic implicit frames to use additional features whose separation logic counterparts are already well-understood by the separation logic community.

6.2 FUTURE WORK

Obviously, all **non-supported language constructs** of Chalice, i.e. the delta between the subset of Chalice as defined in Figure 1 and the existing Chalice language, are to be considered as future work, e.g. channels or class and method refinement.

Likewise, all **incompletenesses** of Syxc enumerated in Section 5.3, especially the heap lookup incompleteness described in Section 5.3.1, and all **known issues** from Section 5.4, e.g. old-expressions in monitor invariants, are considered to be future work.

As stated in Section 3.12, Syxc currently does not check **function termination**, which is crucial for the well-definedness of assertions and thus crucial for the soundness of our verification technique. Function termination, however, is an orthogonal problem which has already been addressed, for example in [SJP10]. On a related note, Syxc also does not try to prove method termination, i.e. it currently does not consider **total correctness**.

Regarding Syxc there are also some implementational details that could be improved. The most worthwhile probably is the **interaction with Z3**, which is currently done on a command-line basis and not via an API. Moreover, term symbols, including functions, are currently extracted from the state and emitted to Z3 each time a proof obligation has to be discharged. When Z3 terminates its state is reset by a stack-like pop operation, which is why all symbols have to be redeclared all over again. It would be much more efficient to push/pop Z3 states only when the symbolic execution branches and to only emit new symbols once when they are created.

Another possible modification of the implementation that might gain some runtime performance is the change from a continuation-passing-based to an iterative implementation. Currently, the continuation-passing style is achieved by intensive use of mutual **recursion**, with the well-known danger of stack overflows. However, an iterative style probably makes it much harder to understand and maintain the implementation. Another alternative might be the continuations introduced to Scala by [RMO09]. Replacing immutable with **mutable data structures** to represent the symbolic state might also gain performance, again with the possible disadvantage of obfuscating the implementation.

The encoding of **mu-fields** and the **holds-function** might also be an aspect of future work. Since they are currently part of the path conditions, each update to them results in an additional forall-quantifed term, thereby increasing the number

[5] http://people.cs.kuleuven.be/~bart.jacobs/verifast/

of assumptions the prover has to cope with. Encoding mu-fields and the holds-function as elements of the heap only might thus yield a significant increase in performance.

A related improvement, which presumably is much easier to realise, is **preserving types** when encoding the state on the level of Z3. References, mu-values and sequences are currently all encoded as symbols of type `Int`. Consequently, all corresponding axioms range over integers, which could result in unnecessary instantiations of quantified axioms and thus could lower performance.

In Section 3.4 we state an axiom inferring **object distinctness** from fractional access permissions. It is presumably possible to add such distinctness assumptions when producing access assertions, thereby rendering the axiom superfluous.

It is claimed [SJP09b, SJP10] that symbolic-execution-based verifiers make it easier for developers to determine the reason for failing assertions and thus facilitate **debugging**, when compared to vcg-based verifiers. We have not investigated in this matter, but it would be interesting to see how a suitable assessment schema for such a purpose would look like, and how Chalice and Syxc perform with respect to it. A related improvement of Syxc would be to include the taken execution branches when reporting error messages, which should already facilitate debugging a lot.

Last but not least, our work has not yet been proven to be sound. Syxc extensive test suite hopefully establishes some trust in the verifier, but a **soundness proof** as provided by Smans, Jacobs and Piessens for their work[6] has still do be done.

[6] `http://people.cs.kuleuven.be/~jan.smans/soundness.pdf`

[Ode10] Martin Odersky. *The Scala Language Specification: Version 2.8*. Nov. 2010. URL: http://www.scala-lang.org/docu/files/ScalaReference.pdf. (Cit. on p. 1).

[LM09] K. Rustan M. Leino and Peter Müller. 'A Basis for Verifying Multi-threaded Programs'. In: *ESOP*. 2009, pp. 378–393. (Cit. on pp. 1, 3, 21, 55).

[Kin76] James C. King. 'Symbolic execution and program testing'. In: *Commun. ACM* 19 (7 1976), pp. 385–394. ISSN: 0001-0782. DOI: http://doi.acm.org/10.1145/360248.360252. URL: http://doi.acm.org/10.1145/360248.360252. (Cit. on p. 1).

[BCO05] Josh Berdine, Cristiano Calcagno and Peter W. O'Hearn. 'Symbolic Execution with Separation Logic'. In: *APLAS*. Ed. by Kwangkeun Yi. Vol. 3780. Lecture Notes in Computer Science. Springer, 2005, pp. 52–68. ISBN: 3-540-29735-9. (Cit. on pp. 1, 55).

[SJP10] Jan Smans, Bart Jacobs and Frank Piessens. 'Heap-Dependent Expressions in Separation Logic'. In: *FMOODS/FORTE*. Ed. by John Hatcliff and Elena Zucca. Vol. 6117. Lecture Notes in Computer Science. Springer, 2010, pp. 170–185. ISBN: 978-3-642-13463-0. (Cit. on pp. 1, 2, 9, 13, 18, 55–57).

[SJP09a] Jan Smans, Bart Jacobs and Frank Piessens. 'Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic'. In: *ECOOP*. Ed. by Sophia Drossopoulou. Vol. 5653. Lecture Notes in Computer Science. Springer, 2009, pp. 148–172. ISBN: 978-3-642-03012-3. (Cit. on pp. 2, 55).

[Boy03] John Boyland. 'Checking Interference with Fractional Permissions'. In: *SAS*. Ed. by Radhia Cousot. Vol. 2694. Lecture Notes in Computer Science. Springer, 2003, pp. 55–72. ISBN: 3-540-40325-6. (Cit. on pp. 2, 55).

[Bar+05] Michael Barnett et al. 'Boogie: A Modular Reusable Verifier for Object-Oriented Programs'. In: *FMCO*. Ed. by Frank S. de Boer et al. Vol. 4111. Lecture Notes in Computer Science. Springer, 2005, pp. 364–387. ISBN: 3-540-36749-7. (Cit. on p. 2).

[Lei08] K. Rustan M. Leino. *This is Boogie 2. (Draft)*. June 2008. URL: http://research.microsoft.com/en-us/um/people/leino/papers/krml178.pdf. (Cit. on p. 2).

[MB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. 'Z3: An Efficient SMT Solver'. In: *TACAS*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. ISBN: 978-3-540-78799-0. (Cit. on p. 2).

[Lei10] K. Rustan M. Leino. 'Verifying Concurrent Programs with Chalice'. In: *VMCAI*. Ed. by Gilles Barthe and Manuel V. Hermenegildo. Vol. 5944. Lecture Notes in Computer Science. Springer, 2010, p. 2. ISBN: 978-3-642-11318-5. (Cit. on pp. 3, 26, 55).

[Dij71] Edsger W. Dijkstra. 'Hierarchical Ordering of Sequential Processes'. In: *Acta Informatica* 1 (1971), pp. 115–138. (Cit. on p. 5).

[FW08] Daniel P. Friedman and Mitchell Wand. *Essentials of programming languages (3. ed.)* MIT Press, 2008. ISBN: 978-0-262-06279-4. (Cit. on p. 18).

[Bru93] Kim B. Bruce. 'A Paradigmatic Object-Oriented Programming Language: Design, Static Typing and Semantics'. In: *Journal of Functional Programming* 4 (1993), pp. 127–206. (Cit. on p. 37).

[SJP09b] Jan Smans, Bart Jacobs and Frank Piessens. *Symbolic Execution for Implicit Dynamic Frames*. Tech. rep. Katholieke Universiteit Leuven, Belgium, 2009. URL: `http://people.cs.kuleuven.be/~jan.smans/oopsla09.pdf`. (Cit. on pp. 44, 55, 57).

[LMS10] K. Rustan M. Leino, Peter Müller and Jan Smans. 'Deadlock-Free Channels and Locks'. In: *ESOP*. Ed. by Andrew D. Gordon. Vol. 6012. Lecture Notes in Computer Science. Springer, 2010, pp. 407–426. ISBN: 978-3-642-11956-9. (Cit. on p. 55).

[DP08] Dino Distefano and Matthew J. Parkinson. 'jStar: towards practical verification for java'. In: *OOPSLA*. Ed. by Gail E. Harris. ACM, 2008, pp. 213–226. ISBN: 978-1-60558-215-3. (Cit. on p. 56).

[JP08] Bart Jacobs and Frank Piessens. *The VeriFast program verifier*. Technical Report CW-520. Department of Computer Science, Katholieke Universiteit Leuven, Aug. 2008. URL: `http://www.cs.kuleuven.be/~bartj/verifast/verifast.pdf`. (Cit. on p. 56).

[PS11] M. J. Parkinson and A. J. Summers. 'The Relationship Between Separation Logic and Implicit Dynamic Frames'. In: *European Symposium on Programming (ESOP)*. to appear. 2011. (Cit. on p. 56).

[RMO09] Tiark Rompf, Ingo Maier and Martin Odersky. 'Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform'. In: *ICFP*. Ed. by Graham Hutton and Andrew P. Tolmach. ACM, 2009, pp. 317–328. ISBN: 978-1-60558-332-7. (Cit. on p. 56).

Appendix

The appendix contains code snippets, usually fragments of Syxc's code, illustrating claims made or information given in the report, and it is referenced at the corresponding positions in the report.

MyType

Listing 26: A snippet illustrating MyType type parameters

```scala
/*
 * Works when using MyType
 */

abstract class A[S <: A[S]] {
  def merge(a: S): S
}

case class A1(a: Int) extends A[A1] {
  def merge(other: A1) = A1(a + other.a)
}

case class A2(a: Int, b: Int) extends A[A2] {
  def merge(other: A2) = A2(a * other.a, b * other.b)
}

def aclient[S <: A[S]](x: S, y: S) = x.merge(y)

/*
 * Type problems without use of MyType
 */

abstract class V {
  def merge(a: V): V
}

case class V1(a: Int) extends V {
  // def merge(other: V1) = V1(a + other.a)
    /* Error: Wrong argument type */

  def merge(other: V) = V1(a + other.a)
    /* Error: a is not a member of V */
}

// case class V2(a: Int, b: Int) extends V { ... }
  /* Same problems */

// def vclient(x: V, y: V) = x.merge(y)
  /* x and y might not be compatible */
```

Syxc's components

Listing 27: State classes

```
trait Permission[P <: Permission[P]] {
  def +(perm: P): P
  def -(perm: P): P
  def *(perm: P): P
}

trait Chunk {
  def rcvr: Term
  def id: String
}

trait Store[S <: Store[S]] {
  def apply(key: Variable): Term
  def updated(key: Variable, value: Term): S
  def +(kv: (Variable, Term)): S
}

trait Heap[S <: Heap[S]] {
  def findChunk(rcvr: Term, id: String): Option[Chunk]
  def +(chunk: Chunk): S
  def -(chunk: Chunk): S
}

trait PathConditions[S <: PathConditions[S]] {
  def contains(t: Term): Boolean
  def +(term: Term): S
}

trait HasStore[ST <: Store[ST], S <: HasStore[ST, S]] {
  def ?: ST
  def \(?: ST): S
  def \+(v: Variable, t: Term): S
}

trait HasHeaps[H <: Heap[H], S <: HasHeaps[H, S]] {
  def h: H
  def g: H
  def \(h: H, g: H): S
  def \+(c: Chunk): S
  def \-(c: Chunk): S
}

trait HasPathConditions[PC <: PathConditions[PC],
                        S <: HasPathConditions[PC, S]] {
  def p: PC
  def \(p: PC): S
  def \+(t: Term): S
}

trait State[ST <: Store[ST], H <: Heap[H],
            PC <: PathConditions[PC],
            S <: State[ST, H, PC, S]]
    extends HasStore[ST, S]
    with HasHeaps[H, S]
    with HasPathConditions[PC, S] {

  def \(?: ST = ?, h: H = h, g: H = g, p: PC = p): S
}
```

Listing 28: Symbolic execution classes

```
trait Evaluator[P <: Permission[P], ST <: Store[ST],
                H <: Heap[H], PC <: PathConditions[PC],
                S <: State[ST, H, PC, S]] {

  def evals(s: S, es: List[Expression], m: Message,
            Q: (S, List[Term]) => VerificationResult)
            : VerificationResult

  def eval(s: S, e: Expression, m: Message,
           Q: (S, Term) => VerificationResult)
           : VerificationResult

  def evalp(s: S, p: FractionalPermission, m: Message,
            Q: (S, P) => VerificationResult)
            : VerificationResult

  def evall(lit: syxc.ast.Literal): terms.Literal
  def evallm(lm: syxc.ast.LockMode): terms.LockMode
}

trait Producer[P <: Permission[P], ST <: Store[ST],
               H <: Heap[H], PC <: PathConditions[PC],
               S <: State[ST, H, PC, S]] {

  def produce(s: S, s: Term, p: P, f: Expression,
              m: Message, Q: S => VerificationResult)
              : VerificationResult
}

trait Consumer[P <: Permission[P], ST <: Store[ST],
               H <: Heap[H], PC <: PathConditions[PC],
               S <: State[ST, H, PC, S]] {

  def consume(s: S, p: P, f: Expression, m: Message,
              Q: (S, Term) => VerificationResult)
              : VerificationResult
}

trait Executor[ST <: Store[ST], H <: Heap[H],
               PC <: PathConditions[PC],
               S <: State[ST, H, PC, S]] {

  def exec(s: S, stmts: List[Statement], m: Message,
           Q: S => VerificationResult)
           : VerificationResult

  def exec(s: S, stmt: Statement, m: Message,
           Q: S => VerificationResult)
           : VerificationResult
}
```