Permission-Based Verification of Subclassing and Traits

Master Thesis
Andres Bühlmann
January 30, 2013
Abstract

Despite the fact that object-oriented languages are well established in general, it is still a challenging task to support subtyping and inheritance in automated program verifiers, especially in the context of permission logics. The same applies even more to traits, which are a means for fine-grained code reuse promoted by programming languages such as Scala. Traits have only recently gained attention in the automated program verification community and are not yet supported in an automated program verifier.

In this thesis we adapt the concept of abstract predicate families developed by Parkinson and Bierman in the context of separation logic to the context of implicit dynamic frames. Based on this, we develop an algorithm for the automated verification of Chalice programs. Chalice is a Microsoft Research language based on implicit dynamic frames that is object-based, supports fork-join concurrency, monitors, fractional permissions, abstract predicates, pure functions and has been extended, as part of this work, with support for subtyping and inheritance. We have implemented and tested the developed algorithm in the automated program verifier Syxc based on symbolic execution.

In line with the long-term goal of Syxc to verify Scala programs, we extend our algorithm in order to verify traits. We show in particular how to verify traits in the presence of late-bound super calls, which are similar to dynamically dispatched calls.
# Contents

<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Overview</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Chalice</td>
<td>1</td>
</tr>
<tr>
<td>1.3 Syxc</td>
<td>1</td>
</tr>
<tr>
<td>1.4 Proceeding</td>
<td>2</td>
</tr>
<tr>
<td>2 Background</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Implicit Dynamic Frames</td>
<td>3</td>
</tr>
<tr>
<td>2.2 Chalice</td>
<td>4</td>
</tr>
<tr>
<td>2.2.1 Introductory Example</td>
<td>4</td>
</tr>
<tr>
<td>2.2.2 Language Constructs</td>
<td>4</td>
</tr>
<tr>
<td>2.2.3 Syntax</td>
<td>10</td>
</tr>
<tr>
<td>2.3 Program Verification and Subtype Polymorphism</td>
<td>10</td>
</tr>
<tr>
<td>2.3.1 The Naive Verification Approach</td>
<td>11</td>
</tr>
<tr>
<td>2.3.2 Supertype Abstraction</td>
<td>12</td>
</tr>
<tr>
<td>2.3.3 Enforcing Behavioural Subtyping</td>
<td>12</td>
</tr>
<tr>
<td>3 Subclassing Support for Chalice</td>
<td>15</td>
</tr>
<tr>
<td>3.1 Motivating Example</td>
<td>15</td>
</tr>
<tr>
<td>3.2 Definition of Subclassing</td>
<td>16</td>
</tr>
<tr>
<td>3.3 Abstract Families</td>
<td>16</td>
</tr>
<tr>
<td>3.3.1 Abstract Predicates</td>
<td>16</td>
</tr>
<tr>
<td>3.3.2 Abstract Predicate Families</td>
<td>18</td>
</tr>
<tr>
<td>3.3.3 Abstract Function</td>
<td>20</td>
</tr>
<tr>
<td>3.3.4 Abstract Function Families</td>
<td>20</td>
</tr>
<tr>
<td>3.4 Separating Implementation, Interface and Type</td>
<td>21</td>
</tr>
<tr>
<td>3.4.1 Code Specification</td>
<td>22</td>
</tr>
<tr>
<td>3.4.2 Motivating the Distinction of Specifications</td>
<td>23</td>
</tr>
<tr>
<td>3.4.3 Implementation Specification</td>
<td>23</td>
</tr>
<tr>
<td>3.4.4 Interface Specification</td>
<td>25</td>
</tr>
<tr>
<td>3.4.5 Type Specification</td>
<td>27</td>
</tr>
<tr>
<td>3.4.6 Transformations of Specifications</td>
<td>28</td>
</tr>
<tr>
<td>3.5 Language Changes and Their Semantics</td>
<td>31</td>
</tr>
<tr>
<td>3.5.1 Methods and Functions</td>
<td>31</td>
</tr>
</tbody>
</table>
3.5.2 Constructors ........................................... 32
3.5.3 Abstract Families in Chalice ............................... 34
3.5.4 Final Classes ........................................... 35
3.5.5 Extending a Class ......................................... 35
3.5.6 Overriding Members ....................................... 36
3.5.7 Respecifying Methods and Functions ......................... 36
3.5.8 Inheriting Members ....................................... 38
3.5.9 Modules .................................................. 38
3.5.10 Deriving the Verification Specifications ....................... 38
3.5.11 Calls and Function Applications ......................... 41
3.5.12 Monitor Invariants ...................................... 44
3.6 Syntax ..................................................... 46

4 Symbolic execution ........................................ 49
4.1 Outline .................................................... 49
4.2 Symbolic Execution in Syxc ................................. 50
4.3 Formalism .................................................. 50
4.4 Specification Refinement ..................................... 50
4.4.1 Definition ............................................... 51
4.4.2 Example for Specification Refinement ....................... 52
4.4.3 Transitivity ............................................. 53
4.4.4 Automation ............................................. 54
4.4.5 Incompleteness Issue .................................... 54
4.5 Supporting Subclasing ....................................... 56
4.6 Supporting Subtyping ....................................... 57
4.6.1 Behavioural Subtyping .................................... 58
4.6.2 Type Specification Conformance ......................... 58
4.6.3 Supertype Abstraction .................................... 59
4.7 Supporting Inheritance ...................................... 60
4.7.1 Interface Specification Conformance ....................... 61
4.8 Adapting the Symbolic Execution Rules ....................... 61
4.8.1 Dynamic Class .......................................... 61
4.8.2 Abstract Predicate Families ............................ 61
4.8.3 Dynamically Dispatched Member Invocations .......... 63
4.8.4 Fork and Join .......................................... 66
4.8.5 Constructors .......................................... 66
4.8.6 Object Sharing ......................................... 68

5 Supporting Subclassing in Syxc ................................ 73
5.1 General Remarks on The Implementation .................... 73
5.2 Software-Architectural Changes in Syxc ..................... 74
5.3 New Representation of Predicate Chunks ..................... 74
5.4 Dealing with Abstract Predicates ............................ 75
5.4.1 Inheriting a Predicate by Aggregation .................... 75
5.4.2 Inheriting Predicate Bodies ............................. 75
5.4.3 Comparing the Notions of Predicate Inheritance ........ 75
5.5 Automating the Proof Obligations ............................ 76
5.5.1 Type Specification Conformance .......................... 76
5.5.2 Interface Specification Conformance ....................... 77
5.5.3 Behavioural Subtyping .................................. 83
5.5.4 Incompleteness Arising from Abstract Predicates ......... 83
5.6 Monitor Invariants ........................................... 85
5.6.1 Monitor Invariants and Subclassing ...................... 85
5.6.2 Solution Approaches .................................... 86
5.6.3 Soundness of the Implemented Solution Approach ....... 88
5.6.4 Sharing an Object without Knowing its Dynamic Type ... 89
6 Traits .................................................................... 91
6.1 Scala Traits ....................................................... 91
6.2 Related Concepts ............................................. 91
6.3 Motivating Example ........................................... 92
   6.3.1 Traits Enriching the Interface .......................... 92
   6.3.2 Traits Introducing Side-Effects ......................... 93
6.4 Adding Traits to Chalice ..................................... 95
   6.4.1 Contract Definition ..................................... 95
   6.4.2 Trait Definition .......................................... 96
   6.4.3 Class Definition ......................................... 97
   6.4.4 Anonymous Mixin Compositions ....................... 98
   6.4.5 Templates ................................................ 98
   6.4.6 Linearization Order .................................... 98
   6.4.7 Late Bound Super Member Invocations ................ 100
   6.4.8 Deviation from Scala Traits ............................. 100
   6.4.9 Definitions ............................................... 101
   6.4.10 Abstract Families and Traits ......................... 102
   6.4.11 Inheritance of Members ................................. 102
6.5 Verification Goal ............................................. 102
6.6 Deriving the Verification Specifications ..................... 103
   6.6.1 Deriving the Implementation Specification ............. 103
   6.6.2 Deriving the Interface Specification ................. 104
   6.6.3 Deriving the Type Specification from the Contract .. 105
6.7 Adapting presented proof obligations ....................... 105
   6.7.1 Type Specification Conformance ....................... 105
   6.7.2 Behavioural Subtyping .................................. 106
   6.7.3 Interface Specification Conformance ................... 107
6.8 Dealing with Late Bound Super Invocations ................. 107
   6.8.1 Static Superclass Abstraction .......................... 107
   6.8.2 Static Specifications .................................... 108
   6.8.3 Derivation of the Static Specification ................. 110
   6.8.4 Proof Obligations ...................................... 111
   6.8.5 Static Superclass Abstraction ......................... 114
   6.8.6 Limitation of Static Superclass Abstraction ......... 115
6.9 Automatizing the Proof Obligations ......................... 116
   6.9.1 Behavioural Subtyping and Behaviour Preservation ... 116
   6.9.2 Type Specification Conformance ....................... 117
   6.9.3 Static Specification Conformance ..................... 118
   6.9.4 Interface Conformance ................................ 119
1.1 Overview

Syxc is an automatic program verifier based on symbolic execution which currently supports the class-based programming language Chalice. However, there is currently neither support for subtyping nor for code-reuse. As the long-term goal of Syxc is to become an automatic program verifier for the Scala programming language [18], verification support for subtyping and code-reuse is important.

The goal of this master’s thesis is to add support for subclassing and traits to the programming language Chalice and to develop and implement the required verification support in the program verifier Syxc.

1.2 Chalice

Chalice [14, 13] is the name of an experimental programming language and also of a static program verifier based on verification condition generation. Both are developed by ourselves in collaboration with Microsoft Research. In this work we will restrict the discussion to the programming language Chalice.

The programming language Chalice was designed having specification and verification of concurrent programs in mind. It is object based, supports fork-join concurrency, predicates, fine-grained locking, fractional permissions, monitors, pure functions as well as the annotation of pre- and post conditions.

However, Chalice currently lacks support for standard and well established programming language features such as subtyping and inheritance.

1.3 Syxc

Syxc is a program verifier for a subset of the Chalice language and is based on symbolic execution. It was developed as part of a master’s thesis [24] to explore the advantages of a program verifier based on symbolic execution for verifying
Chalice programs. Recently, it could be reported [9] that the performance of an approach based on symbolic execution is roughly twice as fast as the initial approach of the Chalice program verifier based on verification condition generation.

1.4 Proceeding

In chapter 2 we introduce the reader to the programming language Chalice as it was prior to this work and briefly discuss some theoretical concepts needed. Chapter 3 presents first the key concepts needed to support subclassing and then details how the programming language Chalice was extended. The underlying theory of the verification approach is presented in chapter 4, which serves as a basis for the correctness argument of our implementation as presented in chapter 5. In chapter 6 we outline how the verification of traits can be supported using the concepts presented in chapter 4. Lastly, we conclude our work in chapter 7 and present ideas for the future development of Syxc.
This chapter introduces the concepts necessary for understanding the discourse of the following work. The reader is expected to be familiar with object oriented programming languages and to have a general understanding of program verification and specification.

2.1 Implicit Dynamic Frames

When verifying a program in a modular way, the implementation of a method or function can not be accessed in general. Thus, the specifications of methods and functions are required to provide an upper bound on the set of heap locations accessed by the implementation. If this information was not available in a modular verification setting, one would have to assume that each method call affects all heap locations. Clearly, this would not allow to verify meaningful programs.

The set of memory locations accessible by a method is called the methods frame. Several methodologies on how to specify the frame of a method have been proposed. The one relevant for this work is called implicit dynamic frames [27] which is based on the dynamic frames [8] methodology developed earlier.

The building block of the implicit dynamic frame approach are access permissions. With each heap location an access permission is associated. A method or function can only access heap locations for which they require the corresponding permissions in their precondition. Thus, the frame of a method is given implicitly by its precondition.

The implicit dynamic frames methodology works nicely with fractional permissions [3]. The idea of fractional permissions is that permissions can be split. For read-access on a heap location only a fraction of the corresponding permission is required. For write-access full permission is required. This allows more flexible specification and is a prerequisite for concurrent read access on heap locations.

In the following we will present the programming language Chalice which uses the implicit dynamic frame methodology in combination with fractional permissions.
2.2 Chalice

Due to the different aspects of program verification that were explored during the last years, the Chalice language has grown quite large. This work will only consider a self-containing subset of the language, namely the subset that is supported by Syxsc. On top of this subset we will then add our own language extensions. An introduction to Chalice can be found in [14, 13] or on the website of Microsoft Research [23].

In the following we introduce the reader to the programming language Chalice and motivate some of the language constructs.

2.2.1 Introductory Example

In listing 2.1 we give an introductory example of a Chalice program. The shown language concepts get introduced one after each other in the remainder of this chapter.

Before continuing we like to point out that classes belong to a module, that we distinguish methods and functions (side-effect-free) and that preconditions and postconditions contain access permissions (for instance the access permission acc(val) for the val field).

```
class Cell module CellLibrary {
  var val : int

  function get(): int
    requires acc(val)
    ensures true
  { val }

  method set(v: int)
    requires acc(val)
    ensures acc(val) && val == v
  { val := v }

  method add(v: int)
    requires acc(val)
    ensures acc(val) && val == old(val) + v
  { val := val + v }
}
```

Listing 2.1: Introductory example without any abstraction mechanism used.

2.2.2 Language Constructs

In the following we describe the language constructs relevant for the further understanding. The language constructs and their semantics are described as they were prior to this work. Where possible we already outline in this chapter what changes are to come.
2.2. CHALICE

Modules

A chalice program consists of modules containing classes. They serve as a mean to partition the program into independent pieces such that interface- and specification-preserving changes in one module do not affect the correctness of other modules. This is one of the mechanisms allowing modular verification of a program.

The consequence of modules with the described semantic is that external clients of a module’s class can never make use of implementation details and can only rely on the specifications given by the methods and functions. This is in contrast to Syxc’s practice of using the function bodies during verification as long as the function definition and function application are within the same module. Using the function bodies during verification has two advantages. Firstly, it allows the programmer to write and verify programs using unspecified implementation details. Secondly, it does not force functions to repeat their body in the specification in case this would be the only meaningful specification (for instance classical ‘getters’).

Classes

Classes are to be understood as in other object oriented languages. The syntax and semantics of classes are to some extent similar to the programming language Java. However, Chalice does neither support inheritance nor subtyping. One key-consequence of this is that all method calls and function applications can be bound statically. Clearly, this will change when we add support for subclassing.

Classes consist of fields, methods and functions, predicates and monitor invariants described in the following.

Methods

While functions are pure, methods are in general not. The effect of methods is described by their precondition, postcondition and lockchange expression.

In listing 2.1 we give an example of a method specified with elementary elements only. The $\text{acc}(\text{val})$ expression denotes read-write-permissions for the $\text{val}$ field. The $\text{old}(\text{val})$ expression denotes the value of the $\text{val}$ field prior to the execution of the method.

Mechanisms for data abstraction

When looking at listing 2.1 and in particular at the given specification of the add method, we observe that the specification reveals implementation details.

Firstly, we reveal that the internal value is stored in a field named ‘val’ as this is the only access permission we require by $\text{acc}(\text{val})$. Secondly, we reveal the same information in the postcondition $\text{val} == \text{old}(\text{val}) + v$. 
CHAPTER 2. BACKGROUND

Beside being not nice from a information-hiding perspective it has an additional drawback. Should we ever rename the field or require additional access-permissions (i.e. to update a cached value), the specification will change and force reverification of all dependent classes and modules.

To circumvent this problem it was proposed [13, 27] to use abstract predicates and abstract functions. Abstract predicates abstract over the access permissions and functions over field-dependent pure expressions. We restate in listing 2.2 the same example as in listing 2.1 using these abstraction mechanisms. The predicate valid abstracts over the access permissions for the val field and the get-function abstracts over the val field itself. Using these two abstraction mechanisms, a client can not infer how the class stores the value and what memory locations are affected exactly by a call to add. Nevertheless, the client (and the verifier) is still able to apply the same reasoning techniques as presented in the section 2.1 on implicit dynamic frames. In particular, the abstract predicates can be used to frame method- and function invocations.

With respect to the following listing, we would like to point out that the valid expression inside the pre- and postconditions takes the role of the access permissions. More precisely, the definition of the valid predicate reveals that valid predicate abstracts over the concrete access permission acc(val). The abstract predicates, functions, the fold and unfold statements as well as the unfolding expression will be discussed in the following in more depth.

```
0 class Cell module CellLibrary { 
1 var val : int 
2 
3 predicate valid { acc(val) } 
4 
5 function get(): int 
6 requires valid 
7 ensures true 
8 { unfolding valid in val } 
9 
10 method set(v: int) 
11 requires valid 
12 ensures valid && get() == v 
13 { 
14 unfold valid 
15 val := v 
16 fold valid 
17 } 
18 
19 method add(v: int) 
20 requires valid 
21 ensures valid && get() == old(get()) + v 
22 { 
23 unfold valid 
24 val := val + v 
25 fold valid 
26 } 
27 }

Listing 2.2: Introductory example with abstraction mechanisms used. The same example was given in listing 2.1 without abstraction mechanisms.
```
Abstract Predicates

Referring to listing 2.2 and using the notions as in [13], the predicate valid can be seen in an abstract view (i.e. folded view) which means we do not see the predicates definition. This is the view clients of a class have in general.

On the other hand, we can see the predicate in a concrete view (i.e. unfolded view) which means we see the definition of the predicate. This is the view needed for body verification. In order to change between the two views, the programmer has to use the fold and unfold statements as well as the unfolding expression. These two statements and the unfolding expression will be denoted ghost statements in the following.

Abstract predicates may not only abstract over access permissions but in addition also over pure expressions. Recalling the last example, we could restrict the stored value to be positive and then change the valid predicate’s body to acc(val) && val > 0.

Given that the body of a predicate is allowed to contain pure expressions, going from the abstract view to the concrete view can be understood as trading the predicates name against the body of the predicate (we assume the body). The other way can then be understood as trading the body against the predicates name (assert the body of the predicate and revoking the permissions).

Functions

In contrast to methods, functions are pure. This implies that they neither can produce nor lose access permissions. Therefore, the access permissions mentioned in the precondition of a function are automatically returned after a function application. More precisely, the access permissions mentioned in the precondition of a function are implicitly contained in the postcondition as well. An example showing that the postcondition of a function does not explicitly return access permissions is given by the get function in listings 2.1 and 2.2.

With respect to functions as a mechanism for abstraction, the programmer does not need to provide ghost statements to change between the concrete view on the function body and the abstract view on the function name as it is the case for abstract predicates. The exchange of the views happens automatically when a function application gets evaluated by the verifier. When a function application gets evaluated, first the function’s precondition is checked to hold and then the body of the function is evaluated. In order to break cycles in the function evaluation (e.g due to a recursive function), the function body only gets evaluated for the first time and for all latter function applications only the function’s postcondition is used.

As Chalice does neither support inheritance nor subtyping, the body of the function is statically known and can be accessed by the verifier given that the module boundaries are respected. Table 2.1 summarizes when the body of a function is evaluated (‘Evaluated’) and when only the specification (‘Specification’) but not
the body is used. The distinction between a syntactical ‘this’ receiver of the function application and an other receiver is motivated by the further development in this work.

<table>
<thead>
<tr>
<th>Definition of function and application are...</th>
<th>within same module</th>
<th>in different modules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receiver is this</td>
<td>Evaluated</td>
<td>Impossible</td>
</tr>
<tr>
<td>Receiver is other than this</td>
<td>Evaluated</td>
<td>Specification</td>
</tr>
</tbody>
</table>

Table 2.1: Evaluation of function applications. The table is valid for function applications appearing in specifications as well as in method-, function-, predicate-bodies, where-clause of channels as well as in monitor invariants.

Before we continue we would like to point out that the evaluation of function applications in specifications is necessary in some cases in order to establish the relation between the model (being the abstract predicates and functions) and the effective implementation (being predicate bodies and fields). An example can be given using the set method of listing 2.2. There, the postcondition \( \text{get()} == v \) needs to be established by the verifier. However, without knowing the equality \( \text{get()} == \text{val} \) the verifier is not able to establish the postcondition. This is the reason why we need to evaluate the function get by which the needed equality gets revealed.

Channels

Channels are an unbounded message buffer and allow data to be passed between concurrent threads in a deadlock-free manner. We give a short introduction and refer to [15] for a full discussion of channels.

Channels come with a non-blocking send and a receive operation that may block if there is currently no message in the channel. In order to avoid a thread from deadlocking due to a receive operation on an empty channel the concept of credits was introduced. A credit is always associated with a thread and a channel.

A thread is only allowed to execute a receive operation if it has at least one credit for the channel. The existence of this credit implies that there is at least one other thread having a send obligation (modelled by a negative credit) on this channel. Positive credits may be discarded, negative credits not. The execution of a receive statement reduces the available credit by one, a send increases the credit by one (reduces the number of messages to send).

A ‘where’ clause in the channel definition allows to define a so called message invariant. Not only does the message invariant constrain the messages transferred over the channel but also does it serve as a mean for transferring permissions over the channel.

```
channel NonNegativeIntChannel(val: int) where val >= 0

class Program {
    method main()
    {
```
2.2. CHALICE

```java
var ch : NonNegativeIntChannel := new NonNegativeIntChannel
call doSend(ch, 3)
call res := doReceive(ch)
assert res >= 0
}

method doSend(ch: NonNegativeIntChannel, value: int )
requires ch != null
requires credit(ch, -1)
requires value >= 0
{
    send ch(value)
}

method doReceive(ch: NonNegativeIntChannel) returns (val : int )
requires ch != null
requires credit(ch, 1)
requires rd(ch.mu) & & waitlevel << ch.mu
ensures val >= 0 // Channel invariant
{
    receive val := ch
}
```

Listing 2.3: Example demonstrating the use of channels (without concurrency).

Sharing Objects

Objects can be shared between threads. In order to guarantee mutual exclusion monitors are used. Monitor invariants associated with classes define the state in which an instance is allowed to be shared and serve as a mean of transferring permissions between threads. For a full introduction to sharing of objects see [14]. In the following we give a summary only.

Initially, an object is thread local and can be shared using the `share` statement. After sharing it, threads can acquire the object using the `acquire` statement and release it again using the `release` statement. Finally, an object can be unshared using the `unshare` statement resulting in the object being thread local again.

Whenever an object is shared or released the monitor invariant has to hold (consider it to be implicitly part of the corresponding statements precondition) and when an object gets acquired the monitor invariant is guaranteed to hold (consider it to be implicitly part of the corresponding statements postcondition).

```java
class SharedCell {
    var val : int

    // The monitor invariants states that
    // only a non-negative cell can be shared
    invariant valid & & get() >= 0

    predicate valid { acc(val) }

    function get(): int
        requires valid
```
**CHAPTER 2. BACKGROUND**

```java
ensures true
{ unfolding valid in val }
method set(v: int)
  requires valid
  ensures valid && get() == v
{
  unfold valid
  val := v
  fold valid
}

class Program {
  method main()
  requires true
  ensures true
  {
  var cell: SharedCell := new SharedCell
  fold cell.valid
  call cell.set(2)
  share cell
  // cell might be accessed by other threads
  acquire cell
  assert cell.get() >= 0
  call cell.set(3)
  release cell
  }
}
```

**Listing 2.4:** Example demonstrating sharing of objects.

### 2.2.3 Syntax

Figure 2.1 summarizes the syntax prior to this work and will serve as a basis to show how the language got extended as a result of this work. Overlining such as $\bar{e}$, $\bar{s}$ represents repetition, $n$ an integer literal, $x$ a local variable, $f$ a field, $m$ a method, $p$ a pure function, $V$ a predicate, $T$ a type and $C$ a class. Angle brackets, $[\ldots]$, are used to indicate that a syntactical element is optional.

### 2.3 Program Verification and Subtype Polymorphism

Let in the following discussion the term *member* be a synonym for methods and functions. When verifying a program without support for subtype polymorphism, the static type of the receiver of a method call or function application uniquely determines the invoked member. Thus, the exact specification of the invoked member is known statically. This is the situation with which the Chalice verifier had to deal prior to this work.

As soon as support for subtype polymorphism (and with it dynamically dispatched calls) gets added, it is in general no longer possible to statically determine
the exact member being executed. Now, the exact specification of the executed member depends on the dynamic type of the receiver. Given that the program was type checked, we can infer that the dynamic type must be a subtype of the static type of the receiver. But without further restrictions on the subtype relation we do not have more information available for verification. This leaves us with the problem that a single member invocation could be bound to an arbitrary implementation in a subtype of the receiver’s static type. In the following we outline the limitations of a naive approach and present the state-of-the-art solution.

### 2.3.1 The Naive Verification Approach

A strait forward solution is to verify each method call and function application against all possible implementations in the subtypes.

However, this neither allows modular verification nor is it particularly efficient. First of all, the introduction of a new subtype would require us to reverify all

Figure 2.1: The subset of Chalice supported by Syxc
parts of the program which use a supertype of the new subtype. This is certainly not modular. Secondly, a program could only be verified in a closed world setting (i.e. all subtypes are known). Thinking of library code, this is not how software is written in practice.

### 2.3.2 Supertype Abstraction

A better solution is to enforce some notion of *behavioural subtyping*. If each subtype is required to behave as all its supertypes then it suffices to verify dynamically dispatched member invocations against the specification of the static type of the receiver.

This verification approach is called *supertype abstraction* [12, 10, 11] as the static type of the receiver (the supertype) abstracts over all its subtypes.

There are several notions of behavioural subtyping. The notion of America [1] which has been extended by Liskov and Wing [16] is commonly known as *Liskov’s substitution principle* or *strong behavioural subtyping*. The notion of strong behavioural subtyping can best be described as requiring a subtype to weaken the supertype’s preconditon and to strengthen the supertype’s postcondition. The resulting pair of implications is sometimes called the *standard pair of implications*. However, this notion is stronger than necessary and therefore some authors use another notion of *weak behavioural subtyping* [11, 5]. The notions differ in what method postconditions need to hold in the case where a subtypes precondition holds but the supertypes precondition does not hold.

### 2.3.3 Enforcing Behavioural Subtyping

While supertype abstraction solves the problem of modular verification for dynamically bound member invocations, it raises the question of how to enforce or attest behavioural subtyping. Several propositions for enforcing or attesting behavioural subtyping exist.

#### Specification Inheritance

Dhara and Leavens [5] propose an approach called *specification inheritance* which guarantees by construction that subtypes are a behavioural subtype. The idea is that the subtype’s method/function specification inherits the supertype’s specification. Thus, the subtypes implementation is verified against the generated specification consisting of the inherited specification and the subtypes own specification. Thereby, an implementation is guaranteed to satisfy its own specification as well as all specifications of its counterparts in superclasses.

#### Reverification against Specifications in Superclass

Another approach was proposed by Poetzsch-Heffter and Müller [21]. Their idea is to show that the members of a new subtype satisfy the specification of their
counterparts in the all direct superclasses. This can be done for instance by re-verifying the implementation in the subtype against the superclass’s specifications.

**Refinement Notion on Specifications**

Leavens and Naumann [11] focus on the specifications and use the notion of refinement to enforce behavioural subtyping. That is, for enforcing behavioural subtyping one has to show that a subtypes specification refines the supertypes specification. The advantage of this approach is clearly that one does not need to verify a body against multiple specifications.
The goal of this chapter is to introduce the new language constructs added to Chalice and to describe language constructs that have changed in their semantics.

We will refer to ‘noSubclassing-Chalice’ as the version of Chalice prior to this work and with ‘subclassing-Chalice’ we refer to the version we describe in the following. If we refer to ‘Chalice’ it gets clear from the context which version we mean.

Before we discuss the new language features we have to introduce abstract families consisting of abstract predicate families and abstract function families. This is needed to understand the new semantics of predicates and function applications in specifications.

In addition to abstract families, we will introduce the concepts of implementation specification, interface specification and type specification which are needed to understand how we deal with inheritance dynamically dispatched member invocations.

We start with a motivating example giving a first impression of the new language constructs.

### 3.1 Motivating Example

Listing 3.1 gives an example of a subclassing-Chalice program. We will often refer to this example as the *running example* or *the motivating example* in the following.

In class `Recell` we override the method `set` as well as the predicate `valid` of the superclass and we inherit the function `get` and methods `add` and `addAlternative`. Furthermore, the two methods `add` and `addAlternative` give an example of methods with and without a *dynamic* modifier, respectively, demonstrating how their implementation differ in general. We denote methods with a *dynamic* modifier external methods and methods without this modifier as internal methods. Internal and external methods differ in how the given specification is interpreted.

In comparison to noSubclassing-Chalice, the following listing contains expressions of the form `valid<Cell>`. This corresponds to indexing of predicate families.
whereby \texttt{Cell} is the index and \texttt{valid} the name of the predicate family. The index can be understood as to select the class from which the body of the predicate \texttt{valid} is used. That is, \texttt{valid<Cell>} refers to the body of the predicate \texttt{valid} given in the \texttt{Cell} class and \texttt{valid<Recell>} refers to the body of the predicate given in the \texttt{Recell} class.

### 3.2 Definition of Subclassing

The exact notion of subclassing will be detailed in the following sections. We only give a rough overview here.

The support of subclassing we add to Chalice can be compared to the programming languages Scala and Java. Classes may extend each other, thereby inheriting members and establishing a subtype relationship. In contrast to Scala and Java, subtypes have to be behavioural subtypes in Chalice. Calls are dynamically dispatched and except for calls on super no statically bound calls are supported.

The syntax and semantics of subclassing-Chalice has been kept as close as possible to Scala’s syntax and semantics.

### 3.3 Abstract Families

Abstract predicate families were introduce by Parkinson and Bierman [19] in the context of separation logic. Abstract predicates in separation logic serve as a mean for abstraction and were adapted by Chalice (see 2.2.2). Abstract predicate families serve as a mean to relate abstract predicates in different (sub-) classes which is a necessity for attesting behavioural subtyping.

In remainder of this section, we will transfer abstract predicate families from a separation logic context to our implicit dynamic frames context and give a formalization of abstract predicates and abstract predicate families. The formalization given is chosen such that it resembles the syntax of Chalice where appropriate. Furthermore, we extend the idea of abstract predicate families to abstract functions and abstract function families for which we give a formalization as well.

#### 3.3.1 Abstract Predicates

**Definition 3.1** An abstract predicate \texttt{rcvr.V<C>} refers to the predicate \texttt{V} defined in class \texttt{C}. As abstract predicates are member of classes, they are associated with an instance \texttt{rcvr} of the defining class or one of its subclasses. Furthermore, abstract predicates define a body \texttt{B} being an expression in which the only free variable is \texttt{this} being bound to the receiver \texttt{rcvr} of the abstract predicate. Thus, a full definition of an abstract predicate is given by \texttt{rcvr.V<C>} := \texttt{B}

When it is clear from the context what the receiver of the abstract predicate is or when it is not necessary to mention it explicitly, we will simply omit the receiver.
3.3. ABSTRACT FAMILIES

```
0  class Cell {
1      var val : int
2      predicate valid { acc(val) }
3
4      // ... constructors omitted
5
6      function get(): int
7          requires valid
8          ensures true
9          { unfolding valid<Cell> in val }
10
11      method set(v: int)
12          requires valid
13          ensures valid && get() == v
14          {
15              unfold valid<Cell>
16              val := v
17              fold valid<Cell>
18          }
19
20      method dynamic add(v: int)
21          requires valid
22          ensures valid && get() == old(get()) + v
23          {
24              var currentVal := get()
25              call set(currentVal + v)
26          }
27
28      method addAlternative(v: int)
29          requires valid
30          ensures valid && get() == old(get()) + v
31          {
32              unfold valid<Cell>
33              val := val + v
34              fold valid<Cell>
35          }
36
37  }
38
39  class Recell extends Cell {
40      var bak : int
41      override predicate valid { this.valid<Cell> && acc(bak) }
42
43      // ... constructors omitted
44
45      function getBak(): int
46          requires valid
47          ensures true
48          { unfolding valid<Recell> in bak }
49
50      override method set(v: int)
51          requires valid
52          ensures valid && get() == v
53          ensures getBak() == old(get())
54          {
55              unfold valid<Recell>
56              bak := super.get()
57              call super.set(v)
58              fold valid<Recell>
59          }
60  }
```

Listing 3.1: Motivating example involving method override, predicate override and inheritance. The full example including the constructors can be found in appendix A.1.
of the abstract predicate. That is, instead of \( e.\mathcal{V}<\mathcal{C}> \) we simply write \( \mathcal{V}<\mathcal{C}> \). We will mostly use this shorthand when the receiver of the abstract predicate is this.

Recall (section 2.2.2), that abstract predicates exist either in an abstract view (we consider them to be a symbol and do not see their body) or a concrete view (we see only the body of the predicate but do not have it as a symbol). An abstract predicate can not exist in the abstract view and concrete view at the same time and the views can be exchanged using fold and unfold statements.

### 3.3.2 Abstract Predicate Families

The purpose of abstract predicate families is to abstract over a set of related abstract predicates. Assume we want to provide a precondition for the following method declaration inside the Cell class of our running example given in listing 3.1.

```java
method swap(cell: Cell)
```

In noSubclassing-Chalice, a meaningful precondition would be `cell.valid<Cell>`. We use the index Cell as in noSubclassing-Chalice cell has to be an instance of class Cell. However, in subclassing-Chalice, we do not know the class of the instance cell statically as it could be an arbitrary subclass. In particular, the abstract predicate we require could depend on the statically unknown class of the instance cell. Abstract predicate families allow us to provide a meaningful precondition for swap method in subclassing-Chalice. For instance, the precondition could be `cell.valid<dynamic>`, referring to the abstract predicate of the dynamic class of the instance cell. We start with the definition of abstract predicate families.

**Definition 3.2** An abstract predicate family \( r_{\text{cvr}}.\mathcal{V} \) denotes the abstract predicate family \( \mathcal{V} \) with receiver \( r_{\text{cvr}} \). An abstract predicate family is the set of abstract predicates that refer to the same abstract predicate \( \mathcal{V} \) and have the same receiver \( r_{\text{cvr}} \).

In order to provide an example based on our motivating example given in listing 3.1, let `recell` be an instance of class Recell. For the abstract predicate family `recell.valid` we have that

```java
recell.valid = \{ recell.valid<Cell>, recell.valid<Recell> \}
```

In a modular verification setting we do not know all subclasses of a particular class in general. Therefore, the exact set of abstract predicates contained in an abstract predicate family can only be determined statically if we know the class from which the receiver has been instantiated (e.g. we know that `recell` is of class Recell in the above example). However, not knowing the exact set of abstract predicates in an abstract predicate family does not pose a problem as the exact set itself is never of interest to us.

### Indexing Abstract Predicate Families

As abstract predicates are uniquely identified by their name per class, we can use a class to index an abstract predicate family. If we understand \( \diamond \) as an indexing
operation then indexing an abstract predicate family $rcvr.V$ becomes syntactically equivalent with referring to a specific abstract predicate $rcvr.V\langle C \rangle$. This motivates the syntactical choice for representing abstract predicates. We denote $C$ as the abstract predicate family index or simply index. For a simpler treatment of abstract predicate families we require the following property.

**Required property 3.3** Let $D$ be a subclass of an arbitrary class $C$. If $C$ defines an abstract predicate then $D$ has to define an abstract predicate with the same name.

The above rule ensures that once a class introduces an abstract predicate family, all subclasses will define an abstract predicate belonging to that family.

**Indexing with the ’dynamic’ Symbol**

Abstract predicate families give us a mean to abstract over the defining class of an abstract predicate (We can for instance express that there is an abstract predicate of unknown index satisfying some requirement). This is exactly what we need for our verification approach based on supertype abstraction and is also required for attesting behavioural subtyping. In both cases we will use abstract predicate families to abstract over the dynamic class of an instance. In order to emphasize that we are not interested in any abstract predicate but in a specific one we introduce a special symbol which can be used as abstract predicate family index.

**Definition 3.4** An abstract predicate family $rcvr.V$ indexed by the special symbol dynamic, that is $rcvr.V\langle \text{dynamic} \rangle$, expresses that the abstract predicate $rcvr.V\langle D \rangle$ is meant where $D$ is the class from which the receiver $rcvr$ has been instantiated from. Furthermore, an abstract predicate indexed with the dynamic symbol does only have an abstract view.

The dynamic symbol allows us to index an abstract predicate family with a statically unknown class. This is a trick allowing us to use a special abstract predicate in places where we would have to use an abstract predicate family. As this facilitates the discussion in the following and emphasizes the role of the abstract predicate family we will not use abstract predicate families any more and only use abstract predicates indexed with the dynamic symbol.

**Definition 3.5** We denote the index of an abstract predicate concrete if the index denotes a class (i.e it is not the dynamic symbol).

We denote the index of an abstract predicate dynamic if its index is the dynamic symbol.

**Abstract- and Concrete View**

Not surprisingly, abstract predicates with a concrete index have a concrete view whereas abstract predicates with a dynamic index do not as they represent an abstract predicate family. We give the rule for exchanging abstract view and concrete view.

**Rule 3.6** The rules for exchanging abstract view and concrete view are as follows:
Given the abstract view of an abstract predicate with a concrete index, the concrete view can only be accessed using the ghost statement \texttt{unfold} and the ghost expression \texttt{unfolding}.

Given the abstract view of an abstract predicate with a dynamic index, the concrete view can not be accessed.

Given the concrete view of an abstract predicate, the abstract view can only be accessed using the ghost statement \texttt{fold}.

Rule 3.6 merely restates the same rules as presented in [13] taking abstract predicate families into account.

3.3.3 Abstract Function

The introduction of abstract functions was motivated in [13]. As their purpose of existence is very similar to abstract predicates it seems reasonable to introduce them in an analogous way. In the following we give definitions related to abstract functions.

\textbf{Definition 3.7} An abstract function \texttt{rcvr.p<C>(a_1,a_2,\ldots,a_n)} refers to the abstract function \textit{p} defined in class \textit{C} with parameter list \textit{a_1,a_2,\ldots,a_n}. As abstract functions are member of classes, they are associated with an instance \texttt{rcvr} of the defining class or one of its subclasses. Furthermore, abstract functions define a body \textit{B} being a pure expression in which the only free variables are \texttt{this} and the functions parameters given by the parameter list \textit{a_1,a_2,\ldots,a_n}. Thus, a full definition of an abstract function is given by \\
\texttt{rcvr.name<C>(pars) := B}

As abstract predicates, abstract functions have an \textit{abstract view} (we consider them to be a symbol and do not see their body) or a \textit{concrete view} (we see only the body of the function but do not have it as a symbol). In contrast to abstract predicates, the views can not be exchanged manually by the programmer and only get exchanged by the verifier.

3.3.4 Abstract Function Families

We discussed the evaluation of functions in 2.2.2. With the addition of subclassing, function applications can not be evaluated any more without statically knowing the class of the receiver as function applications are dynamically bound in general. An abstract function allows us to refer to the implementation of a function in a particular class. Abstract families allow us to refer to a particular abstract function from a set of related abstract functions (i.e we can refer to the abstract function defined in the dynamic class of the receiver which is not known statically in general).

Due to the similarity of abstract predicate families and abstract function families we do not further discuss the definitions and only state them. We only ask the reader to observe rule 3.12 which does not correspond directly to the rule given for abstract predicates.
Definition 3.8 An abstract function family $\text{rcvr}.p(a_1, a_2, \ldots, a_n)$ refers to the abstract function family $p$ with receiver $\text{rcvr}$ and parameter list $a_1, a_2, \ldots, a_n$. An abstract function family is the set of abstract functions that refer to the same abstract function $p$ and have the same receiver $\text{rcvr}$.

Required property 3.9 Let $D$ be a subclass of an arbitrary class $C$. If $C$ defines an abstract function then $D$ has to define an abstract function with the same name.

Definition 3.10 An abstract function family $\text{rcvr}.p(a_1, a_2, \ldots, a_n)$ indexed by the special symbol $\text{dynamic}$, that is $\text{rcvr}.p<\text{dynamic}>(a_1, a_2, \ldots, a_n)$, expresses that the abstract function $\text{rcvr}.p<D>(a_1, a_2, \ldots, a_n)$ is meant where $D$ is the class from which the receiver $\text{rcvr}$ has been instantiated from.

Furthermore, an abstract function indexed with the $\text{dynamic}$ symbol does only have an abstract view.

Definition 3.11 We denote the index of an abstract function concrete if the index denotes a class (i.e. it is not the $\text{dynamic}$ symbol).

We denote the index of an abstract function dynamic if its index is the $\text{dynamic}$ symbol.

Rule 3.12 The concrete view of an abstract function can only be accessed if all of the following holds

- The index of the abstract function is concrete.
- The function application is from within the same module as the definition of the function. That is, the abstract function’s index refers to a class belonging to the same module as the function application.
- The function’s precondition holds.

Observe that the above rule takes into account module boundaries whereas rule 3.6 does not. This is for historical reasons and is supposed to change in the future such that module boundaries are respected by abstract predicates as well.

### 3.4 Separating Implementation, Interface and Type

In this section, we will motivate the distinction of four kinds of specifications. We denote the specification given by the programmer the code specification. The additional three specifications, which we will denote verification specifications, will be generated automatically from the code specification. The three verification specifications are the implementation specification, interface specification and the type specification. We shortly describe for what purpose we will use the different specifications in order to give an intuition. The type specification will be used to attest behavioural subtyping and to verify dynamically dispatched member invocations. We will used the interface specification to verify statically bound member invocations and for attesting that inheriting a member from the superclass is sound. Furthermore, we will use the implementation specification for body verification and use it to abstract over the concrete body.

The generated verification specifications differ only in the indices to the abstract families from the code specification, being syntactically identical otherwise. A
formal definition of the transformations used for the generation will be given in section 3.5.10. The verification specifications are tightly related to each other by proof obligations (i.e. for behavioural subtyping) forcing the chosen indices for abstract families in one verification specification to be compatible with the indices in the other two verification specifications. However, for the moment we restrict us on getting an intuition for each of the verification specifications.

The idea of having more than one specification per class is not new. Parkinson and Bierman [20] propose a dynamic- and static specification (corresponding to our type- and interface specification). Poetzsch-Heffter and Müller [22] distinguish specifications (i.e annotations) for dynamically dispatched (i.e virtual methods) and statically bound methods which corresponds again to our type- and interface specification.

The third kind of specification we introduced - implementation specification - is motivated by code reuse and allows to abstract over the reused code. Parkinson and Bierman [20] do not explicitly have such an implementation specification but derive an implementation specification on demand. In our terms but their approach, the inherited member’s implementation specification equals the interface specification of the source class of the member.

We will first discuss the code specifications given by the programmer, then motivate the distinction of the verification specifications and afterwards discuss each verification specification in detail.

### 3.4.1 Code Specification

As the code specification gets written directly by the programmer, writing it should be as convenient and intuitive as possible. As the focus of this work is not on designing a specification language, we simply assume that convenient means less specification in terms of required letters and intuitive means that reasonable and simple default rules apply whenever the programmer does not specify something explicitly.

This is one reason why we allow the programmer to omit indices to abstract families in the specifications. It requires fewer letters, avoids repetition and does not force the programmer to be overly precise.
3.4.2 Motivating the Distinction of Specifications

There is another reason to allow programmers to omit indices to abstract families. The specification of a member is used to verify the body, to attest behavioural subtyping and to verify that inheriting a member is sound. For each of these aspects, the programmer potentially has to index the abstract families differently. That is, if a programmer wants to provide indices in general, he has to decide on a particular aspect for which to provide the indices. We give an example based on the motivating example in listing 3.1.

Taking the body verification perspective and looking at the postcondition of the Cell.set method, we would index the abstract families with \texttt{Cell}. That is, we have 
\begin{verbatim}
this.valid<Cell> && this.get<Cell>() == v.
\end{verbatim}

Doing the analogous for the postcondition of the set method in the Recell class, we have
\begin{verbatim}
this.valid<Recell> && this.get<Recell>() == v.
\end{verbatim}

The indices have been chosen to be the equivalent to the enclosing class. This can be motivated when looking at the corresponding method bodies and in particular the unfold- and fold statements which use the same indices. Furthermore, choosing the indices to be equal to the enclosing class can be motivated by the requirement for behavioural subtyping. We therefore take the behavioural subtyping perspective next.

When taking the behavioural subtyping perspective, we would index the abstract families in the specification of the set method of the classes Cell and Recell with \texttt{dynamic}. That is, we have
\begin{verbatim}
this.valid<dynamic> && this.get<dynamic>() == v
\end{verbatim}

Indexing the abstract families with \texttt{dynamic} rather than with a concrete index reflects that we do not want to force all subclasses to use the exact same abstract predicates and abstract functions. In particular, this allows to use different indices in the body verification perspective as done for the postcondition of the set method in this example. Observe, that for an instance of class \texttt{Cell}, the body verification perspective and the behavioural subtyping perspective get equivalent as the \texttt{dynamic} symbol refers to \texttt{Cell} in this case. The analogous happens for the Recell class.

Having observed that we need in some situations to index abstract families differently depending on whether we take the perspective of body verification, behavioural subtyping verification or inheritance verification, this motivates why we generate for each of these perspectives an own specification.

3.4.3 Implementation Specification

The implementation specification of a member is the specification that is used to verify the method’s body and is generated automatically from the code specification of the method. The implementation specification of a class contains for every
member of the class a specification independently of whether the member is inherited or defined explicitly in the class. Members are inherited together with their implementation specification.

In the following two listings we provide the implementation specification of the Cell and the Recell class of our running example (listing 3.1). The aspect to observe in the listing for the Cell class is that the specification of add has been indexed with dynamic while the specification of addAlternative has been indexed with Cell. This difference stems from the modifier dynamic which is provided for the add method. The purpose of the dynamic modifier is exactly to provide a mean to index the abstract families with dynamic instead of the enclosing class which is the default behaviour. The default behaviour is motivated by the fact that it aligns with the requirement for behavioural subtyping (e.g. for an instance of class Cell the dynamic symbol is equivalent to Cell).

 Generated implementation specification of class Cell {

  function get(): int
    requires this.valid<Cell>
    ensures true

  method set(v: int)  
    requires this.valid<Cell>
    ensures this.valid<Cell>
    ensures this.get<Cell>() == v

  method add(v: int)  
    requires this.valid<dynamic>    
    ensures this.valid<dynamic>
    ensures this.get<dynamic>() == old(this.get<dynamic>()) + v

  method addAlternative(v: int)  
    requires this.valid<Cell>
    ensures this.valid<Cell>
    ensures this.get<Cell>() == old(this.get<Cell>()) + v


Listing 3.2: Part of the implementation specification of class Cell. The full example can be found in appendix A.1.1.

For the implementation specification of the Recell class, two aspects are to observe. First, the specification of the set method, which overrides the Cell.set method, has been indexed with Recell. This corresponds again to the default behaviour. Secondly, it is to observe that the inherited members have the same specification as in the superclass.

 Generated implementation specification of class Recell {

  function getBak(): int
    requires this.valid<Recell>
    ensures true

  method set(v: int)
    requires this.valid<Recell>
    ensures this.valid<Recell>
    ensures this.get<Recell>() == v
3.4. Separating Implementation, Interface and Type

Ensures this.getBak<Recell>() == old(this.get<Recell>())

// Inherited from Cell
function get(): int
  requires this.valid<Cell>
  ensures true

// Inherited from Cell
method add(v: int)
  requires this.valid<dynamic>
  ensures this.get<dynamic>() == old(this.get<dynamic>()) + v

// Inherited from Cell
method addAlternative(v: int)
  requires this.valid<Cell>
  ensures this.valid<Cell>
  ensures this.get<Cell>() == old(this.get<Cell>()) + v

Listing 3.3: Part of the implementation specification of class Recell. The full example can be found in appendix A.1.1.

3.4.4 Interface Specification

As explained above, when a class $D$ inherits a member from a superclass, the member is inherited together with its implementation specification. However, this may not be the intention of the programmer in terms of the class interface of $D$ as the inherited specification may contain abstract predicates and abstract functions indexed by the superclass.

We give an example. Let recell be an instance of class Recell. In listing 3.3 we observe the postcondition given for the inherited Recell.addAlternative method. The postcondition is equivalent to the following (we omitted the this receiver due to space constraints)

ensures valid<Cell> && get<Cell>() == old(get<Cell>()) + v

Focusing on the function application of get in the postcondition, it specifies the effect a call to addAlternative has on the get function of the Cell class. However, if we invoke addAlternative on recell we are interested in what effect the method call has on the Recell.get, and not its effect on the get function of the Cell class. Thus, we need to provide a new specification for the class Recell aligning the specifications of inherited members with the inheriting class. We call this specification the interface specification. The postcondition of addAlternative in the interface specification of class Recell is given by

ensures valid<Recell> && get<Recell>() == old(get<Recell>()) + v

If a client uses the postcondition of the interface specification above, then the effect of the method call addAlternative on recell is described in terms of the get function of the Recell class, which is the function a client can inspect itself.
This observation, namely that for inherited members the inherited specification is not suited for clients to reason about member invocations, motivates the introduction of the interface specification. The interface specification gets generated automatically from the code specification and it can be understood as the intended specification of the class. The interface specification is the one that is used by clients to reason about statically bound member invocations (e.g., super calls and constructor invocations).

We use again our running example 3.1 for illustration. The interface specification of the Cell class is identical to its implementation specification. This stems from the fact that the class Cell does not inherit any members. Looking at the interface specification of the Recell class, it is to observe that the specification of get and addAlternative have changed compared to their implementation specifications. This is exactly in order to align the specifications of the inherited members with the inheriting class.

```java
// Generated interface specification of class Cell {
  ...
  function get(): int
    requires this.valid<Cell>
    ensures true

  method set(v: int)
    requires this.valid<Cell>
    ensures this.valid<Cell>
    ensures this.get<Cell>() == v

  method add(v: int)
    requires this.valid<dynamic>
    ensures this.valid<dynamic>
    ensures this.get<dynamic>() == old(this.get<dynamic>()) + v

  method addAlternative(v: int)
    requires this.valid<Cell>
    ensures this.valid<Cell>
    ensures this.get<Cell>() == old(this.get<Cell>()) + v
}

// Generated interface specification of class Recell {
  ...

  function getBak(): int
    requires this.valid<Recell>
    ensures true

  method set(v: int)
    requires this.valid<Recell>
    ensures this.valid<Recell>
    ensures this.get<Recell>() == v
    ensures this.getBak<Recell>() == old(this.get<Recell>())

  function get(): int
    requires this.valid<Recell>
    ensures true

  method add(v: int)
    requires this.valid<dynamic>
    ensures true
```
3.4. SEP ARATING IMPLEMENTATION, INTERFACE AND TYPE

ensures this.valid<dynamic>
ensures this.get<dynamic>() == old(this.get<dynamic>()) + v

method addAlternative(v: int)
  requires this.valid<dynamic>
  ensures this.get<dynamic>() == old(this.get<dynamic>()) + v

Listing 3.4: Part of the interface specification of class Cell and Recell. The full example can be found in appendix A.1.2.

3.4.5 Type Specification

The intention of the type specification is to abstract over at least one but usually many interface specifications. This gets reflected in the specification that the abstract family indices get replaced by the special dynamic symbol.

In the following listing all abstract family indices have been replace by dynamic. In theory this is not necessarily always the case. However, providing a concrete class as abstract predicate family index in the type specification of a method in some class C imposes a strong restriction on all subclasses of C. Subclasses of C will have to use literally the same indices in their specification as C and are not able to use their overridden version of a predicate or function any more.

Listing 3.4 gave an example where a subclass uses the overridden version of a predicate. There, the Recell class provides different indices than the Cell class for the set method which is only possible because the type specification of Cell.set uses the dynamic symbol. If the type specification for the Cell.set method used concrete indices, then the same indices would have to be used by the interface specification of class Recell for the set method, as a consequence of behavioural subtyping.

The following listing gives the type specification of our running example (listing 3.1). Observe that all indices are dynamic. That is, when reasoning about a dynamically dispatched call to a reference r of static type Cell (it could refer to an instance of class Recell for example) we can use the type specification of class Cell. Thereby, the type specification nicely expresses that we do not know exactly with which abstract predicates and abstract functions we deal with, but we know that they correspond to the abstract predicates and abstract functions defined in the dynamic class of the instance referred to by the reference r.
ensures this.valid<dynamic>
ensures this.get<dynamic>() == v
method add(v: int)
  requires this.valid<dynamic>
  ensures this.valid<dynamic>
  ensures this.get<dynamic>() == old(this.get<dynamic>()) + v
method addAlternative(v: int)
  requires this.valid<dynamic>
  ensures this.valid<dynamic>
  ensures this.get<dynamic>() == old(this.get<dynamic>()) + v
}

Generated type specification of class Recell {
  var bak: int
  predicate valid
  function get(): int
    requires this.valid<dynamic>
    ensures true
  function getBak(): int
    requires this.valid<dynamic>
    ensures true
  method set(v: int)
    requires this.valid<dynamic>
    ensures this.valid<dynamic>
    ensures this.get<dynamic>() == v
    ensures this.getBak<dynamic>() == old(this.get<dynamic>())
  method add(v: int)
    requires this.valid<dynamic>
    ensures this.valid<dynamic>
    ensures this.get<dynamic>() == old(this.get<dynamic>()) + v
  method addAlternative(v: int)
    requires this.valid<dynamic>
    ensures this.valid<dynamic>
    ensures this.get<dynamic>() == old(this.get<dynamic>()) + v
}

Listing 3.5: Type specification of the class Cell and Recell. The full example can be found in appendix A.1.3.

The type specification is used whenever the dynamic class of the instance/receiver is not known or shall not be considered. This is the case for verification of dynamically dispatched calls (supertype abstraction) and for attesting behavioural subtyping.

3.4.6 Transformations of Specifications

We present three syntactical transformations of specifications dealing with abstract families in specifications. The transformations are indexing, syntactical indexing.
and syntactical reindexing. The syntactical indexing transformations corresponds
to the syntactical opening presented by Parkinson and Bierman [20].

These transformations will later be used to derive the verification specifications
from the code specifications and will also be involved in proving certain properties
between the verification specifications.

The transformations will be given in a pattern matching style. Thereby, only the
first pattern that matches applies. In order to distinguish symbols and terminals in
the patterns, symbols are in italic and terminals in typewriter font. Furthermore,
we use \( V \) for abstract predicate families and \( p \) for abstract function families.

**Indexing Transformation**

The indexing transformation takes a specification and a class \( C \) as index which is
then used to index all abstract families with \( C \). Note that it leaves abstract families
that are already indexed untouched.

In the following we will not give the full transformations as it only affects abstract
families and leaves the specification otherwise unaffected.

Let \((\text{pre}, \text{post})\) be a specification of a member. The indexing transformation \( I \) is
given by

- \( I((\text{pre}, \text{post}), C) := (I(\text{pre}, C), I(\text{post}, C)) \)
- \( I(e_1 \text{ op } e_2, C) := I(e_1, C) \text{ op } I(e_2, C) \) where \( \text{op} \) is an arbitrary binary operation (e.g. \&\&
  \text{==} > +)
- \( I(\text{op}(e)) := \text{op}(I(e, C)) \) where \( \text{op} \) is an arbitrary unary operation (e.g. !)
- \( I(\text{unfolding pa in e}) := \text{unfolding } I(pa, C) \text{ in } I(e, C) \)
- ...
- \( I(\text{rcvr.V<index>}, C) := \text{rcvr.V<index>} \)
- \( I(\text{rcvr.V}, C) := \text{rcvr.V<C>} \)
- \( I(\text{rcvr.p<index>(args)}, C) := \text{rcvr.p<index>(args)} \)
- \( I(\text{rcvr.p(args)}, C) := \text{rcvr.p<C>(args)} \)

**Syntactical indexing transformation**

The syntactical indexing transformation takes a specification and a class \( C \) as index
which is then used to index all abstract families with \( C \) given that the associated
instance syntactically is identical to this. Instances that are not syntactically identical
to this get indexed with the symbol dynamic. Note that it leaves abstract families
that are already indexed untouched.

Let \((\text{pre}, \text{post})\) be a specification of a member. The syntactical indexing transfor-
mation \( SI \) is given by
The syntactical reindexing transformation associated instance is syntactically identical to
this

\[ SI((pre, post), C) := (SI(pre, C), SI(post, C)) \]

\[ SI(e_1 \text{ op } e_2, C) := SI(e_1, C) \text{ op } SI(e_2, C) \] where \( \text{op} \) is an arbitrary binary operation (e.g. &&, ==, +)

\[ SI(op(e)) := op(SI(e, C)) \] where \( \text{op} \) is an arbitrary unary operation (e.g !)

\[ SI(\text{unfolding } pa \text{ in } e) := \text{unfolding } SI(pa, C) \text{ in } SI(e, C) \]

\[ \ldots \]

\[ SI(rcur.V<index>, C) := rcur.V<index> \]

\[ SI(this.V, C) := this.V<C> \]

\[ SI(rcur.V, C) := rcur.V<\text{dynamic}> \]

\[ SI(rcur.p<\text{index}>(\text{args}), C) := rcur.p<\text{index}>(\text{args}) \]

\[ SI(this.p(\text{args}), C) := this.p<C>(\text{args}) \]

\[ SI(rcur.p(\text{args}), C) := rcur.p<\text{dynamic}>(\text{args}) \]

Syntactical Reindexing Transformation

The syntactical reindexing transformation takes a specification and two classes \( C \) and \( O \). All abstract families indexed by \( O \) are reindexed with \( C \) given that the associated instance is syntactically identical to this. Not that abstract families that are not indexed are unaffected by this transformation.

Let \((pre, post)\) be a specification of a member. The syntactical reindexing transformation \( SR \) is given by

\[ SR((pre, post), C, O) := (SR(pre, C, O), SR(post, C, O)) \]

\[ SR(e_1 \text{ op } e_2, C, O) := SR(e_1, C, O) \text{ op } SI(e_2, C, O) \] where \( \text{op} \) is an arbitrary binary operation (e.g. &&, ==, +)

\[ SR(op(e), C, O) := op(SR(e, C, O)) \] where \( \text{op} \) is an arbitrary unary operation (e.g !)

\[ SR(\text{unfolding } pa \text{ in } e, C, O) := \text{unfolding } SR(pa, C, O) \text{ in } SR(e, C, O) \]

\[ \ldots \]

\[ SR(this.V<0>, C, O) := this.V<C> \]

\[ SR(rcur.V<\text{index}>, C, O) := rcur.V<\text{index}> \]

\[ SR(rcur.V, C, O) := rcur.V \]

\[ SR(this.p<0>(\text{args}), C, O) := this.p<C>(\text{args}) \]

\[ SR(rcur.p<\text{index}>(\text{args}), C, O) := rcur.p<\text{index}>(\text{args}) \]

\[ SR(rcur.p(\text{args}), C, O) := rcur.p(\text{args}) \]
3.5 Language Changes and Their Semantics

This section depicts the language changes and new features added to the programming language Chalice. We will discuss the semantics of the language constructs as they were implemented and discuss the design decisions we have taken.

In subclassing-Chalice the following language constructs have a new or a more specific semantics:

- Distinction between internal and external methods and functions
- Module definition of classes
- Method calls and function applications
- Monitor invariants of classes

The following language concepts were added in subclassing-chalice:

- Constructors
- Inheritance of methods and functions
- Respecification of methods and functions

3.5.1 Methods and Functions

Inside a class, methods and functions are uniquely identified by their name. Thus, a member is within a class uniquely identified by its name. This rules out any notion of method or function overloading.

We distinguish internal and external methods and functions. The two kinds are primarily motivated by the transformation of the code specification to the verification specifications. This will be detailed in section 3.5.10. Imprecisely, the abstract families inside the specification of external members will be indexed by the dynamic symbol whereas the abstract families inside the specification of an internal member will be indexed with the enclosing class.

As a consequence of the different indexing for internal and external members, the implementation of internal members will mostly be limited to field access and super calls. On the other side, the implementations of external members are mostly restricted to dynamically bound member invocations. Thus, their implementation could very-well be implemented externally by a client - which explains the name.

In a subclassing-Chalice program, an external method or functions has a dynamic modifier. If no such modifier is given, the method or function is considered to be internal. We used a dynamic modifier rather than an external modifier as it better reflects its effect on the generated specifications (i.e the chosen indices will be dynamic). We nevertheless use the terms internal and external as the obvious alternative terms being static and dynamic would lead to a confusion with static methods which are something different.

An example of an internal and external method is given in our running example 3.1 by the methods add and addAlternative.
3.5.2 Constructors

Constructors were added newly to subclassing-Chalice and their semantics has been chosen as close as possible to the semantics of constructors in Scala [18].

Primary and Auxiliary Constructors

There are two kinds of constructors in a class: A primary constructor and auxiliary constructors. A class has exactly one primary constructor and may have zero or more auxiliary constructors. If a class does not define a primary constructor a default construct is generated. In order to distinguish primary and auxiliary constructors, primary constructors have a primary modifier. Constructors do not have a dynamic modifier. When referring with the term member to methods, functions and constructors, we consider a constructor to be an internal member.

A primary constructor must invoke the primary constructor of the superclass as the first statement and is not allowed to invoke auxiliary constructors. Auxiliary constructors may call the primary constructor of the same class or other auxiliary constructors of the same class. The call to other constructors has to be the first statement. The only exception is the primary constructor of the special AnyRef class which does not have a superclass and therefore does not have a super constructor call.

To avoid cyclic calls by auxiliary constructors, they are only allowed to call an auxiliary constructor defined ‘earlier’ (textual position in the source file) in the class.

Constructors can not be called by methods or functions. Therefore, constructors can only be invoked explicitly from within other constructors or implicitly in new statements. Furthermore, constructor overloading is not supported. A constructor is uniquely identified by the number of its parameters.

Listing 3.6 provides an example of constructor definitions.

Constructors as a Source of Permissions

Constructors serve as well as a source of permissions. That is, in the primary constructor immediately after the super constructor call but before the following statement, the permissions for all the fields defined in the class (i.e. not including superclass fields) are generated. This is achieved by injecting the new ghost statement InitDeclaredFields during a preprocessing step. The ghost statement InitDeclaredFields can not be used by the programmer directly.

```java
class Cell {
    var val: int
    primary this () requires true
        ensures valid
    {
        call super()
    }
}
3.5. LANGUAGE CHANGES AND THEIR SEMANTICS

Listing 3.6: Example for constructors in a class (extending running example in listing 3.1).

```java
class Cell {
    var val: int

    predicate valid { acc(val) }

    primary this ()
    ensures valid && get() == 0
    {
        call super()
        val := 0
        fold valid<Cell>
    }

    this(v: int)
    ensures valid && get() == v
    {
        call this()
        call set(v)
    }

    // ... other members omitted
}

class Recell extends Cell {
    var bak: int

    override predicate valid { this.valid<Cell> && acc(bak) }

    primary this ()
    ensures valid && get() == 0 && getBak() == 0
    {
        call super()
        bak := 0
        fold valid<Recell>
    }

    this(v: int)
    ensures valid && get() == v && getBak() == 0
    {
        call this()
        call set(v)
    }

    // ... other members omitted
}
```
CHAPTER 3. SUBCLASSING SUPPORT FOR CHALICE

3.4.3 Abstract Families in Chalice

We have introduced abstract families on a theoretical level in section 3.3 and outlined how they can be indexed. While we give the programmer a mean to index abstract predicate families, we do never allow this for abstract function families.

The reason for disallowing a programmer to index abstract function families is that abstract function families indexed by an arbitrary class correspond to statically bound function evaluations (that is we ignore the dynamic class of the receiver and execute the function of the static class). However, this is something we do not want to support except for statically bound calls on super, for which the programmer does not need to index the function families explicitly.

Abstract Predicate Families in Member Bodies

Indexing abstract predicate families is done as presented in our formalism by `rcvr.predicatename<index>`. If the program does not give an index explicitly a default rule applies. We first give an example in the following listing involving indexing and then present the rule for selecting a default index.

We use our running example and provide a new implementation for the method `set`. We have chosen a new implementation in order to illustrate that multiple fold and unfold ghost statements may be required. In particular, we observe that the fold and unfold statements only differ in the indices to the abstract predicate families.

```java
class Recell extends Cell {
    var bak: int

    override predicate valid { this.valid<Cell> && acc(bak) }

    function getBak(): int
        requires valid
```

Special fields such as the `mu` field are treated specially. Its access permissions are created when a class gets instantiated using the `new` statement. A constructor may require access permission to its special fields in its precondition.

The postcondition of a generated primary constructor contains the access permissions of all fields accessible in the class, except for the special fields.

Listing 3.7: Example of a primary constructors where the generation of access permissions is indicated.

```scala
InitDeclaredFields
    // Ghost statement injected by preprocessing step:
    // Creation of access permission to all declared fields.
    // In this example the 'val' field.

    fold valid<Cell>
}

// ... other members omitted
}
```

We have introduced abstract families on a theoretical level in section 3.3 and outlined how they can be indexed. While we give the programmer a mean to index abstract predicate families, we do never allow this for abstract function families.

The reason for disallowing a programmer to index abstract function families is that abstract function families indexed by an arbitrary class correspond to statically bound function evaluations (that is we ignore the dynamic class of the receiver and execute the function of the static class). However, this is something we do not want to support except for statically bound calls on super, for which the programmer does not need to index the function families explicitly.

Abstract Predicate Families in Member Bodies

Indexing abstract predicate families is done as presented in our formalism by `rcvr.predicatename<index>`. If the program does not give an index explicitly a default rule applies. We first give an example in the following listing involving indexing and then present the rule for selecting a default index.

We use our running example and provide a new implementation for the method `set`. We have chosen a new implementation in order to illustrate that multiple fold and unfold ghost statements may be required. In particular, we observe that the fold and unfold statements only differ in the indices to the abstract predicate families.

```java
class Recell extends Cell {
    var bak: int

    override predicate valid { this.valid<Cell> && acc(bak) }

    function getBak(): int
        requires valid
```
3.5. LANGUAGE CHANGES AND THEIR SEMANTICS

```java
ensures true
{ unfolding valid in bak } override method set(v: int)
  requires valid
  ensures valid && get() == v
  ensures getBak() == old(get())
  {
    assert this.valid<Recell> // Given by precondition
    unfold valid<Recell>
    assert valid<Cell> && acc(bak)
    unfold valid<Cell>
    assert acc(bak) && acc(val)
    bak := val
    val := v
    fold valid<Cell>
    fold valid<Recell>
  }
```

Listing 3.8: Alternative implementation of the `set` method of the running example 3.1.

Default Indices for Abstract Predicate Families in Member Bodies

The default index for abstract predicate families in ghost statements and ghost expressions is the static type of the associated instance. That is, if an abstract predicate family `rcvr.V` appears in a ghost statement or ghost expression it gets indexed as `rcvr.V<C>` where `C` is the static type of `rcvr`.

This rule allows to omit the index to abstract predicate families in a lot of cases while it is straightforward for the programmer to deduce the implicit index by himself.

3.5.4 Final Classes

A class with a `final` modifier is denoted a `final class` and is declared as

```
final class C
```

A final class can not be subclassed any further. This implies that having a receiver whose static type is the final class `C`, the receiver’s dynamic class has to be `C` as well.

3.5.5 Extending a Class

A class `D` extends another class `C` using the `extends` keyword by

```
class D extends C
```

Classes can not extend itself and a class `D` can only extend a class `C` that does not itself directly or indirectly extend `D`. Furthermore, the class being extended must not be a final class. If a class does not specify an extends clause the class implicitly extends the special class `AnyRef` known from Scala.
3.5.6 Overriding Members

Overriding Methods and Functions

A subclass can only override a method or function that has been declared in a super class. The overriding member must be of the same kind (method or function) as the overridden member, must have an override modifier, contravariant parameter types and covariant return types.

Hiding a member is not allowed. That is, if a direct or indirect superclass \( C \) contains a member with the same name as a member in the subclass \( D \), then \( D \) must override it.

Overriding Predicates

A subclass can only override a predicate already that has been declared in a super class. The overriding predicate must have an override modifier and can provide an arbitrary body.

Hiding a predicate is not allowed. That is, if a direct or indirect superclass \( C \) contains a predicate with the same name as a predicate in the subclass \( D \), then \( D \) must override it.

3.5.7 Respecifying Methods and Functions

In some cases one may want to override the specification of a superclass member only but not provide a new implementation. We denote this as respecification of a member. We first state the requirements on a respecification and then motivate it using an example.

A subclass can only respecify a method or function that has been declared in a super class. The respecifying member must be of the same kind (method or function) as the respecified member, must have a respecify modifier, invariant parameter types and invariant return types.

Observe in particular that the parameter and return types need to be invariant for which the reason is that otherwise type errors could arise. Within the same module one could allow contravariant parameter types and covariant return types by type-checking them together with the respecified body. However, this will not work past module boundaries as the bodies are not available in this case. For that reason, the implementation currently does not support it at all.

Motivating Support for Respecification

The following listing gives an example based on our running example. Observe the new postcondition given for the \texttt{set} method.
Respecification allows to specify the effect of inherited methods on newly added state in subclasses. To be more precise, respecification allows to specify the effect of the inherited method on new state whose access permissions were added to an abstract predicate belonging to one of the abstract predicate families framing the method. This is exactly what the example above presents.

As it seems, respecification will often be necessary/wished for all inherited members given that the subclass adds new state in the described way. Pretending that the set method was not respecified as in the above example, a call to the set method on an instance of class TwoCell would make the client forget all he knows about the result of get2.

The reader may argue that set2 and get2 should have their own abstract predicate families which is a justified objection to the given example. However, this would reveal to some extent implementation details which may not always be preferred and it may not always be straightforward to separate the state of a class into independent pieces. Thinking of the extreme case where we abstract over each field with a separate abstract predicate family, we would better not use any mean of abstraction.

Given the discussion above, we believe that respecification together with a careful design of the introduced abstract predicate families will provide the best solution. As a last remark we refer to outlook 7.2.2 where we present an idea for automatically respecifying methods.
3.5.8 Inheriting Members

Inheriting methods and functions

A method or function gets inherited from a superclass by a subclass if the subclass does not override or respecify it.

The semantics of inheriting a member \( m \) is equivalent to overriding it with a member invoking \( m \) by a super call and having literally the same code specification. However, it may be the case that additional ghost statements are required. In case the specifications contain predicates within implications, it may even be necessary to execute the ghost statements only under certain conditions.

Inheriting Predicates

A predicate gets inherited from a superclass by a subclass if the subclass does not override it.

Predicates are inherited by aggregation. The semantics of inheriting a predicate \( p \) in a class \( D \) which extends class \( C \) is equivalent to overriding \( p \) in \( D \) with a predicate having the body \( \text{this}.p<\text{C} \) or equivalently \( \text{super}.p \) which refers to the abstract predicate of the superclass.

3.5.9 Modules

The semantics of modules remains as presented in section 2.2.2. In this section we discuss the consequences of modules on subclassing across module boundaries. To give an example, let \( C \) be a class in module \( MC \) and let \( D \) be a subclass of \( C \) in module \( MD \).

Subclassing across module boundaries is permitted. However, the restrictions stemming from module boundaries are enforced as usual. In particular, function applications to a super class in a different module are not evaluated.

3.5.10 Deriving the Verification Specifications

As explained in section 3.4, we distinguish three kinds of specifications used for verification. This section details how we automatically derive the three verification specifications from the code specification.

The main task of the transformations consists in defining the indices for the abstract families for all of the verification specifications. In fact, the verification specifications of a member resulting from its code specification will be syntactically identical up to the indices of the abstract families.

In the following we will refer with the term members to methods, functions and constructors.
3.5. LANGUAGE CHANGES AND THEIR SEMANTICS

Deriving the Implementation Specification from the Code Specification

We have to describe how to derive the implementation specification of an entire class as well as how to derive it for a specific member defined in a class. We denote the transformation described in the following as implementation specification transformation.

We give the rule for a single member first. The implementation specifications of members result from the code specification by providing indices for the abstract families. Otherwise, the two specifications are syntactically identical.

Let $C_m$ denote a code specification of a member $m$ introduced or overridden in class $C$ and let $B_m$ be the implementation specification of $m$. The indices for abstract families (predicates and functions) are selected as follows.

1. If $m$ is an external member, then all indices are dynamic.
   That is $B_m := I(C_m, \text{dynamic})$

2. If $m$ is an internal member, then use the syntactical indexing rule.
   That is $B_m := S I(C_m, C)$

Given all implementation specifications of members we can give the rule for composing the implementation specification of a class as in the following.

1. Let $DM$ be the set of implementation specifications of members introduced or overridden in the class $C$

2. Let $IM$ be the set of implementation specifications of methods and functions introduced or overridden in a direct or indirect superclass but not overridden in class $C$. Imprecisely, this are the implementation specifications of the inherited members.

3. The implementation specification $B$ of class $C$ is given as $B := DM \cup IM$

An implementation specification of our running example was given in listing 3.2. The $\text{add}$ method is an instance of an external method and the $\text{addAlternative}$ method is an instance of an internal method.

Deriving the Interface Specification from the Code Specification

The interface specification of a class $C$ results from the following sequence of operations. We denote this the interface specification transformation.

1. Let $DM$ be the set of code specifications of members introduced, overridden or respecified in the class $C$

2. Let $IM$ be the set of code specifications of methods and functions introduced, overridden or respecified in a direct or indirect superclass but not overridden in class $C$. Imprecisely, this are the code specifications of the inherited members.

3. Let $IT := DM \cup IM$ be the temporary interface specification of class $C$
4. The interface specification $I$ of class $C$ results from applying the following transformation to all specifications in $IT$.

Let $IT_m$ denote a code specification of a member $m$ in $IT$. The indices for abstract families (predicates and functions) in $IT$ are selected as follows.

a) If $m$ is an external member, then all unspecified indices are dynamic.
   That is $I_m := I(IT_m, \text{dynamic})$

b) If $m$ is an internal member, then use the syntactical indexing transformation.
   That is $I_m := SI(IT_m, \text{C})$

We observe that the transformation is similar to the one for implementation specifications. The difference is in the order of the syntactical indexing and inheriting the specification of members. For implementation specifications we first syntactically index the specifications and inherit afterwards. For interface specifications it is the reverse order. The reason is, that for the implementation specification of a class we want to inherit members together with their implementation specifications. However, for the interface specification of a class we want to provide a different specification for inherited members.

An implementation specification of our running example was given in listing 3.4. The get, add and addAlternative methods of class Recell are the most interesting to observe.

**Deriving the Type Specification from the Code Specification**

The type specification of a class $C$ results from the following sequence of operations. We denote this the type specification transformation.

1. Let $DM$ be the set of code specifications of methods and functions introduced, overridden or respecified in the class $C$

2. Let $IM$ be the set of code specifications of methods and functions introduced, overridden or respecified in a direct or indirect superclass but not overridden in class $C$. Imprecisely, this are the code specifications of the inherited members.

3. Let $TT := DM \cup IM$ be the temporary type specification of class $C$

4. The type specification $T$ of class $C$ results after applying the following transformation to all methods and functions in $TT$.

   Let $TT_m$ denote a code specification of a member $m$ in $TT$. The unspecified indices for abstract families (predicates and functions) in $TT$ are selected to be dynamic. That is $T_m := I(TT_m, \text{dynamic})$

Observe that the rule only deviates from the interface specification transformation in how the specification of internal members gets transformed. This reflects the fact that type specification will be used in the verification of dynamically dispatched member invocations.
A type specification of our running example was given in listing 3.5. The `get`, `add` and `addAlternative` methods of class `Recell` are the most interesting to observe.

### 3.5.11 Calls and Function Applications

Function applications (i.e `rcvr.func()`) or calls (i.e `call rcvr.meth()`) were always statically bound in noSubclassing-Chalice and are either statically bound or dynamically bound in subclassing-Chalice. In subclassing-Chalice, member invocations on `super` are statically bound while all other invocations are dynamically bound (i.e dynamically dispatched).

Subclassing-Chalice supports statically bound member invocations on `super` where the invoked members does not necessarily have to be overridden by the calling member. An example for statically bound member invocations on `super` can be found in our running example (listing 3.1) in method `set` of class `Recell`. There we invoke `super.get()` inside the method `set`.

**Statically Bound Member Invocations on Super**

Member invocations on `super` to another member than the overridden one is not supported in languages such as Java or Scala. We decided to support it nevertheless. The reason is that without such invocations we may lose the possibility to access fields of a superclass defined in another module.

The following listing gives an example. Verification will not allow to unfold the predicate `valid` indexed by `C` in function `getPlusOneFails` as the predicate is defined in another module\(^1\). Using the function application `super.get()` we can retrieve the value of the field in the function `getPlusOneSucceeds`. Note, that in both cases no call on `this.get()` is possible as this were a dynamically dispatched function application for which we do not have the necessary access permissions (i.e we would have to use the `dynamic` modifier for the functions).

```plaintext
0 class C module M C {
1     var val: int
2
3     predicate valid { acc(val) }
4
5     function get(): int
6         requires valid
7         { unfolding valid in val }
8 }
9
10 class D extends C module M D {
11     override predicate valid { this.valid<C> }
12
13     function getPlusOneFails(): int
14         requires valid
15}
```

\(^1\)The current implementation will allow it for backward compatibility reasons. But this is to change in the future. As a consequence of the module boundary, verification will nevertheless fail as no relationship between the `val` field and the `get` function can be established (i.e it is an incompleteness).
CHAPTER 3. SUBCLASING SUPPORT FOR CHALICE

ensures result == get() + 1
{
  unfolding valid<D> in unfolding valid<C> in val + 1
}

function getPlusOneSucceeds(): int
requires valid
ensures result == get() + 1
{
  unfolding valid<D> in super.get() + 1
}

Listing 3.10: Accessing fields of superclasses defined in other modules.

Statically Bound Member Invocations in General

Supporting statically bound member invocation on super raises the question why we do not support them on other receivers as well, for example on an instance obj.

The reason why we do no support it is merely simplicity of the language. Observe that the dynamic modifier of members does not influence how a member invocation gets bound (i.e we do not have a similar situation as in C++ with the virtual modifier). Thus, it is the runtime system which decides how a member invocation on obj gets bound. Supporting statically bound member invocations on obj would force a programmer to take into account that clients may actually by-pass the dynamic dispatch mechanism and invoke members that were overridden. Furthermore, it would require another syntax in order to distinguish them from dynamically dispatched invocations. Lastly, as the long term goal of Syxc is to support Scala which does not allow statically bound calls, we do not have to support it.

Coexistence of Statically- and Dynamically Bound Member Invocations

The goal of this subsection is to detail a limitation of our verification approach. Namely, that it is not possible to have dynamically bound member invocations on this together with statically bound member accesses on super or field accesses on this within the same body of a method or function. To be precise, field accesses and dynamically bound member invocations can not coexist within the same method only if the access permissions of the field are contained within an abstract predicate.

To illustrate why accessing dynamically bound members and fields on this is not possible at the same time, we give an alternative implementation of the Cell.add method of listing 3.1.

method addBroken(v: int)
requires this.valid<Cell>
ensures this.valid<Cell>
ensures this.get<Cell>() == old(this.get<Cell>()) + v
{

}
The reason why verification has to fail can best be illustrated using the permissions. The addBroken method requires the permission this.valid<Cell>. The method could have been invoked in two ways. First of all, it could have been invoked from an external client using a dynamically dispatched call. We do not discuss this case any further as it does not produce an interesting example. Secondly, it could have been invoked by a super call. In this case the dynamic class of this does not equal Cell. Therefore, the dynamically dispatched call on set in the body of addBroken would be dispatched to a method in a subclass of Cell. This method may require (and access) memory locations not given by addBroken’s precondition. That is, additional access permissions could be required for the execution of the dynamically bound call.

We give yet another implementation of the add method. Thereby we provide the dynamic modifier to the method declaration. We observe that in contrast to the previous example, the dynamically bound member invocation to get verifies and we are not able to unfold any more - which was the other way round in the previous example.

```java
method dynamic addBroken2(v: int)
  requires this.valid<dynamic>
  ensures this.get<dynamic>() == old(this.get<dynamic>())-> + v
{
  var currentVal := this.get()
  unfold this.valid<Cell>
  // Body verification correctly fails in the above statement.
  // The reason is the following: The access permissions required
  // by the unfold statement is this.valid<Cell> which is not
  // necessarily equivalent to this.valid<dynamic>. In particular, the two will
  // differ if the addBroken method gets called by a super call.
  val := currentVal
  fold this.valid<Cell>
}
```

Listing 3.12: Alternative implementation of the add method of class Cell given in listing 3.1. The abstract families have been index to reflect the implementation specification of the method.

The problem that arises in this situation is that the subclasses of Cell could override the predicate valid and not aggregate the super class’ predicate. In this case,
there is no way in which we could unfold \texttt{this.valid<dynamic>} to derive the required abstract predicate \texttt{this.valid<Cell>} for the execution of the \texttt{unfold} statement. This justifies why the verification has to fail in general.

Having seen the two examples, we observe that independently how we index the abstract predicate families, we can not access fields and invoke dynamically bound members on \texttt{this} within the same body of a method, if the access permissions to the fields are contained in abstract predicates. We do not provide a mean to resolve this limitation in this work. However, one could impose additional restrictions on subclasses and thereby resolve the limitation. This is discussed in the outlook 7.2.3.

### 3.5.12 Monitor Invariants

Monitor invariants were introduced in section 2.2.2 in the context of object sharing. In subclassing-Chalice an object can be shared only if its dynamic class is known. This restriction is unfortunately necessary in order to ensure that the monitor invariant gets established correctly. In order to support subtype polymorphism at least partially, acquiring is possible without knowing the dynamic type of the acquired instance. However, an acquired instance has to be released using the same static type as it has been acquired with.

The semantics of monitor invariants is as follows. Each class can provide a monitor invariant which has to be self-framing as in noSubclassing-Chalice. We denote this the \textit{defined monitor invariant} of a class. The \textit{effective monitor invariant} of a class \(D\) results from combining the \textit{defined monitor invariants} of \(D\) and all of its superclasses using the \&\&-operator. The abstract families in the define defined- and effective monitor invariant are indexed using the symbol \texttt{dynamic}. This is necessary as the client does not know the dynamic class of the instance acquired in general.

When sharing an object, the effective monitor invariant of the object’s dynamic class has to be established. When acquiring an object, the effective monitor invariant of the static class can be assumed. In order to release an object the monitor invariant that was assumed when acquiring the object has to be re-established again. This is ensured and represented by the \texttt{holds(rcvr)} expression in which the static type of \(rcvr\) determines the class at which the \(rcvr\) object has to be released. However, if \texttt{holds(rcvr)} appears in a negative position the static type does not matter. Unsharing an object is possible at an arbitrary static class. If an object does not get unshared at its dynamic class permissions may get lost which will prevent the object from ever been shared again.

Listing 3.13 provides an example of object sharing involving subtype polymorphism.

```java
class Base {
    invariant acc(valid, 50) & getA() >= 0
    var a: int
    predicate valid { acc(a) }
```
3.5. LANGUAGE CHANGES AND THEIR SEMANTICS

```java
    primary this ()
    ensures valid
    {
        call super()
        fold valid
    }

    function getA(): int
    requires rd(valid)
    { unfolding rd(valid) in a }

    method setA(value: int)
    requires valid
    ensures valid && getA() == value
    {
        unfold valid
        a := value
        fold valid
    }

    class SubBase extends Base {
    invariant acc(valid, 50) && getB() >= 0

    var b: int

    override predicate valid { this.valid<Base> && acc(b) }

    primary this ()
    ensures valid
    {
        call super()
        fold valid
    }

    function getB(): int
    requires rd(valid)
    { unfolding rd(valid) in b }

    method setB(value: int)
    requires valid
    ensures valid && getB() == value
    {
        unfold valid
        b := value
        fold valid
    }

    class Program {
        method shareInstance () {
            var sb := new SubBase()
            call sb.setA(2)
            call sb.setB(2)
            share sb
        }

        // Acquire at class 'Base'
        method acquireAndReleaseInstance(c: Base)
```
Listing 3.13: Demonstrate sharing of objects.

3.6 Syntax

Figure 3.2 summarizes the syntax of subclassing-Chalice. Overlining such as $\tilde{e}$, $\tilde{s}$ represents repetition, $n$ an integer literal, $x$ a local variable, $f$ a field, $m$ a method, $p$ a pure function, $V$ a predicate, $T$ a type and $C$ and $D$ a class. Angle brackets, $[..]$, are used to indicate that a syntactical element is optional.
3.6. SYNTAX

\begin{align*}
\text{program} & ::= \overline{tld} \\
tld & ::= \text{cls} | \text{chan} \\
\text{chan} & ::= \text{channel C where } \varphi \\
\text{cls} & ::= [\text{final}] \text{ class D extends C invariant } \varphi \{ \text{mem} \} \\
\text{mem} & ::= \text{var } f:T | \text{pred } | \text{constructor } | \\
& \quad \text{meth } | \text{func } | \text{methRespec } | \text{funcRespec} \\
\text{constructor} & ::= [\text{primary}] \text{ this}(x:T) \\
& \quad \text{requires } \varphi \text{ ensures } \varphi \{ \overline{x} \} \\
\text{meth} & ::= [\text{override}] [\text{dynamic}] \text{ method } m(x:T) \text{ returns } (y:T) \\
& \quad \text{requires } \varphi \text{ ensures } \varphi \text{ lockchange } \overline{x} \{ \overline{x} \} \\
\text{func} & ::= [\text{override}] [\text{dynamic}] \text{ function } p(x:T):T \\
& \quad \text{requires } \varphi \text{ ensures } \varphi \text{ lockchange } \overline{x} \\
\text{methRespec} & ::= \text{respecify } [\text{dynamic}] \text{ method } m(x:T) \text{ returns } (y:T) \\
& \quad \text{requires } \varphi \text{ ensures } \varphi \text{ lockchange } \overline{x} \\
\text{funcRespec} & ::= \text{respecify } [\text{dynamic}] \text{ function } p(x:T):T \\
& \quad \text{requires } \varphi \text{ ensures } \varphi \\
\text{pred} & ::= [\text{override}] \text{ predicate } V<C> \{ \varphi \} \\
\text{cs} & ::= s \mid \text{this}(\overline{x}) \mid \text{super}(\overline{x}) \\
\text{s} & ::= e.f := e | \text{var } x : T | x := e | x := \text{new } C(\overline{x}) | \\
& \quad \text{if}(e) \text{ then } \{ \overline{x} \} \text{ else } \{ \overline{x} \} \text{ call } \overline{x} := e.m(\overline{x}) | \\
& \quad \text{call } \overline{x} := \text{super}.m(\overline{x}) \mid \text{fork } x := e.m(\overline{x}) | \\
& \quad \text{join } \overline{x} := e | \text{fold } p\overline{a} \mid \text{ unfold } p\overline{a} | \\
& \quad \text{share } e \text{ between } \overline{x} \text{ and } \overline{x} \mid \text{ unshare } e \mid \\
& \quad \text{rd } \text{ acquire}(e,lm) \mid \text{rd } \text{ acquire}(e) \mid \\
& \quad \text{rd } \text{ release}(e,lm) \mid \text{rd } \text{ release}(e) \mid \\
& \quad \text{free } e \mid \text{ assert } \varphi \mid \text{ assume } \varphi \mid \\
& \quad \text{while}(e) \text{ invariant } \varphi \text{ lockchange } \overline{x} \{ \overline{x} \} \\
\varphi & ::= \varphi \& & \varphi \mid e \text{ => } \varphi \mid e ? \varphi : \varphi \mid \text{fa } \mid \text{pa } | \text{ e } | \text{ old}(\varphi) \\
\text{e} & ::= e \text{ op } e \mid \text{null } | \text{true } | \text{false } | n \mid \text{lockbottom } | \\
& \quad \text{waitlevel } | e.f | e.p(\overline{x}) | \text{super}.p(\overline{x}) | \\
& \quad \text{unfolding } p\overline{a} \text{ in } e | \text{rd } \text{ holds}(e, lm) \mid \text{rd } \text{ holds}(e) \mid \text{old}(e) \\
\text{op} & ::= \text{aop } | \text{bop } | \text{rop} \\
\text{aop} & ::= + | - | * | / | \% \\
\text{bop} & ::= & & & | | | | | | <= | < | <= | > | >= | << \\
\text{rop} & ::= \text{acc}(e.V, e) \mid \text{acc}(e.V) \mid \text{acc}(e.V, e) \mid \text{acc}(e.V, e) \mid \text{acc}(e.V, e) \mid \text{acc}(e.V, e) \mid \text{acc}(e.V, e) \mid \text{acc}(e.V, e) \mid \text{acc}(e.V) \mid \text{acc}(e.V) \mid \text{acc}(e.V) \mid \text{acc}(e.V) \\
& \quad \text{rd}(e.V, e) \mid \text{rd}(e.V) \mid \text{rd}(e.V, e) \mid \text{rd}(e.V, e) \mid \text{rd}(e.V, e) \mid \text{rd}(e.V, e) \\
& \quad \text{rd}(\text{super}.V, e) \mid \text{rd}(\text{super}.V) \\
\text{fa} & ::= \text{acc}(e.f, e) \mid \text{acc}(e.f) \mid \text{rd}(e.f, e) \mid \text{rd}(e.f) \\
\text{lm} & ::= R | W
\end{align*}

Figure 3.2: The subset of Chalice supported by Syxc including language changes for subclassing support.
With subclassing we mean the combination of inheritance and subtype polymorphism. It is the goal of this work to add support for subclassing to the programming language Chalice as well as to add verification support to Syxc. The subtyping aspect is limited to behavioural subtyping following the notion of specification refinement from Leavens and Naumann [11].

4.1 Outline

There are three major aspects we have to deal with when adding support for subclassing to Syxc:

- **Body verification**
  As discussed in section 2.3 we have to take care of dynamically dispatched member invocations. We deal with this by enforcing behavioural subtyping which allows us to use the approach of supertype abstraction for the verification of dynamically dispatched member invocations.

- **Behavioural subtyping**
  For every subclass we need to attest that it is a behavioural subtype of its supertypes using the notion of specification refinement.

- **Inheritance**
  When inheriting a member of a superclass we need to attest that the member’s implementation satisfies the interface specification of the inheriting class. This takes care of functions and/or predicates which might have been overridden in the inheriting class in an incompatible way.

Furthermore, there are minor aspects we need to deal with:

- **Channels**
  With channels come credit expressions which have to be considered in particular for attesting behavioural subtyping.

- **Monitor invariants of classes**
  Classes allow to specify monitor invariants. The semantics of monitor invariants has to be adapted in order to work with subtype polymorphism.
4.2 Symbolic Execution in Syxc

The following discussion will assume that the reader is familiar with the work of Schwerhoff [24]. We will restate required definitions and concepts from his work but not go into details.

When referring to symbolic execution in the following, the symbolic execution algorithm as presented and implemented by Schwerhoff is meant.

All symbolic execution rules we present in this thesis were formalized in the context of the symbolic execution rules given by Schwerhoff.

4.3 Formalism

We will neither use nor define a full proof system in the following. However, we will use a notation similar to a proof system in order to express the proof obligations we need to discharge using the symbolic execution algorithm. In the following we will assume the basic definitions as given by Schwerhoff.

As in Schwerhoff’s work, let \( T \) be the theory of the symbolic execution engine containing for instance axioms and function symbols.

A symbolic state \( \sigma \) describes the information available at a particular point during symbolic program execution. The state \( \sigma \) gets manipulated by the symbolic execution of statements and is used to symbolically evaluate expressions and assertions. It consists of the symbolic store \( \sigma.\gamma \), the symbolic heap \( \sigma.h \), the symbolic old heap \( \sigma.g \) and the path conditions \( \sigma.\pi \).

We write \( T(\sigma) \vdash \phi \) to express that \( \phi \) follows from the symbolic state \( \sigma \) together with the background theory \( T \). To keep the notation convenient we will use the shorthand \( \sigma \vdash \phi \).

Let \( (A, A') \) denote a specification where \( A \) is the precondition and \( A' \) the postcondition.

Let \( \{A\}m\{A'\} \) denote a Hoare-Triple with a method or function body \( m \) and specification \( (A, A') \).

We write \( T \vdash \{A\}m\{A'\} \) if the implementation \( m \) satisfies the specification \( (A, A') \). That is, if the method- respectively function validation rule as given by Schwerhoff is satisfied.

4.4 Specification Refinement

Following the idea of Leavens and Naumann we introduce specification refinement for functions and methods.
4.4. SPECIFICATION REFINEMENT

4.4.1 Definition

Before we give the definition for specification refinements we need another definition allowing us to abstract over ghost statements and ghost expressions in method and function bodies.

**Definition 4.1** Let $b$ be a method or function body an let $\mathcal{G}$ be a family of body transformations. That is, for any $G \in \mathcal{G}$, $G(b)$ is a transformed body of $b$.

Respecting syntactical rules, the transformations in $\mathcal{G}$ are allowed to

- leave the body untransformed
- inject arbitrary unfold and fold ghost statements
- inject arbitrary unfolding ghost expressions
- inject an if-statement containing unfold and fold ghost statements
- inject an if-then-else expression. Thereby, the body can be duplicated for the 'then' and 'else' branch. In addition, the body may be contained in additional unfolding expressions.

The transformations in $\mathcal{G}$ are not allowed to change the provided body in any other way as the ones specified above.

We observe that the behaviour of a body is unaffected by a body transformation as specified above. The ghost statements and ghost expressions are merely specification annotation and do not affect the behaviour of the implementation. The injection of if-statements and if-then-else-expressions will be needed in order to deal with abstract predicates appearing in the specification in the position of the consequent of an implication (e.g, we can unfold an abstract predicate only if it is indeed present).

**Definition 4.2** A specification $(\{B\}, \{B\}')$ of a method or function refines another specification $(\{A\}, \{A\}')$, formally written as $T \vdash \{A\} \Rightarrow \{A\}'$, if there exists a proof showing that if $T \vdash \{A\} b \{A\}'$ holds for an arbitrary body $b$ then there exists $G \in \mathcal{G}$ such that $T \vdash \{B\} G(b) \{B\}'$ holds.

We have to use the body transformation $G$ in $T \vdash \{B\} G(b) \{B\}'$ as the body $b$ itself would not satisfy $(B, B')$ in general due to potentially missing ghost statements and ghost expressions. Observe that a specification refinement provides as well a proof for the existence of a transformed body $G(b)$ satisfying $(B, B')$, given that there exists a body $b$ satisfying $(A, A')$.

In the following we will always assume that the parameter types and return types of the method/function specification $\{B\} \Rightarrow \{B\}'$ are compatible to the ones in $\{A\} \Rightarrow \{A\}'$. Compatible means to have contravariant parameter types and covariant return types and a of course the same number of parameters and return values.

\footnote{The body would not pass body verification (i.e., not satisfy the specification) due to the missing ghost statements. If there were no ghost-statements in the language, the body would pass the body verification and therefore satisfy the specification $(B, B')$}
CHAPTER 4. SYMBOLIC EXECUTION

Specification Refinements and Type Refinements

We quantify over specification satisfying member implementations in the definition of specification refinements. Observing that behavioural subclasses may override members and specify additional behaviour (e.g. by weakening the precondition), we conclude that the set of satisfying implementations for the specification of the overriding member is a superset of the satisfying implementations for the overridden member’s specification. This explains why the specification of the superclass member refines the specification of the subclass member.

Observe that the notion of specification refinement does not align with the notion of refinements for types. That is, while a behavioural subtype is considered to refine its supertypes, the specification of a method in the subclass gets refined by the specification of the overridden method in the superclass.

4.4.2 Example for Specification Refinement

We give an example for a specification refinement based on the motivating example in listing 3.1 and in particular the addAlternative method inherited by the Recell class. The purpose of the example is to motivate the need for a transformed body in the definition 4.2 of specification refinements.

The addAlternative method has the following implementation and interface specification in class Recell. We as well provide an implementation satisfying the implementation specification provided.

```
0 Implementation specification of Recell.addAlternative {
1     method addAlternative(v: int) {
2         requires this.valid<Cell>
3         ensures this.valid<Cell>
4         ensures this.get<Cell>() == old(this.get<Cell>()) + v
5     }
6 }
7 Interface specification of Recell.addAlternative {
8     method addAlternative(v: int) {
9         requires this.valid<Recell>
10        ensures this.valid<Recell>
11        ensures this.get<Recell>() == old(this.get<Recell>()) + v
12     }
13 }
14 Implementation satisfying the implementation specification {
15     unfold valid<Cell>
16     val := val + v
17     fold valid<Cell>
18 }
```

Listing 4.1: Implementation and interface specification of method addAlternative of class Recell. We as well provide a body satisfying the implementation specification. The definition of the predicates and functions was given in listing 3.1.

Assume we want to show that the interface specification of addAlternative, denoted as \((I, I')\), refines the implementation specification, denoted as \((B, B')\). As we will see later, this corresponds to verifying that the method addAlternative can safely be inherited by the Recell class. Formally, we want to show the following
4.4. SPECIFICATION REFINEMENT

\[ T \vdash \{ B \} \Rightarrow \{ I \} \]

The above specification refinement will be verified without involving an implementation. However, for the sake of the example and as we want to motivate the need for a transformed body, we use an implementation.

Observe that the provided implementation, which we denote as \( b \), satisfies the implementation specification \( (B, B') \) (we need the additional knowledge that \( \text{get} \) returns the \( \text{val} \) field, which is the case and we assume here). That is, \( T \vdash \{ B \} b \{ B' \} \).

The question is how what the a transformed body, \( G(b) \), has to look like in order to satisfy the interface specification \( (I, I') \). The answer is given by the following listing.

```plaintext
0  Transformed implementation satisfying the interface specification {
1    unfold valid<Recell>
2    unfold valid<Cell>
3    val := val + v
4    fold valid<Cell>
5    fold valid<Recell>
6 }
```

**Listing 4.2:** Transformed body satisfying the interface specification. The definition of the predicates was given in listing 3.1.

Observe the additional unfold and fold statement that is required, which corresponds to the effect of the body transformation \( G \). We observe that, \( T \vdash \{ I \} G(b) \{ I' \} \) where \( G(b) \) corresponds to the above implementation.

### 4.4.3 Transitivity

We want to show that specification refinements are transitive given our definition 4.2.

**Theorem 4.3** Specification refinements as given in definition 4.2 are transitive. That is, given \( T \vdash \{ U \} \Rightarrow \{ V \} \) and \( T \vdash \{ V \} \Rightarrow \{ W \} \) it holds that

\[ T \vdash \{ U \} \Rightarrow \{ W \} \]

**Proof sketch** Let \( b \) be an arbitrary body and let \( G_b \) be a body transformer such that \( T \vdash \{ V \} G_b(b) \{ V' \} \) holds, given that \( T \vdash \{ U \} b \{ U' \} \) holds. Note that this is given by \( T \vdash \{ U \} \Rightarrow \{ V \} \). We write \( G_b \) to express that the chosen body transformer depends on the body \( b \). For notational convenience we will drop the subscript in the following, however.

By the definition of specification refinements, \( T \vdash \{ V \} \Rightarrow \{ W \} \) yields that if there exists a proof showing that if \( T \vdash \{ V \} b' \{ V' \} \) holds for an arbitrary body \( b' \) then there exists \( G' \in G \) such that \( T \vdash \{ W \} G'(b') \{ W' \} \) holds.

As the above holds for an arbitrary body \( b' \) satisfying \( \{ V \} b' \{ V' \} \), it holds in particular for a body \( G(b) \) satisfying \( T \vdash \{ V \} G(b) \{ V' \} \), given that \( T \vdash \{ U \} b \{ U' \} \) holds.
As a consequence, $T \vdash \{W\} G'(G(b)) \{W'\}$ holds given that $T \vdash \{U\} b \{U'\}$ holds.

It remains to show that there exists a $G^* \in \mathcal{G}$ such that $G^*(b) = G'(G(b))$. We chose $G^*$ to be the body transformer first proceeding as $G$ and then as $G'$ yielding the desired result.

All together we have that, if $T \vdash \{U\} b \{U'\}$ holds for an arbitrary body $b$ then there exists $G^* \in \mathcal{G}$ such that $T \vdash \{W\} G^*(b) \{W'\}$ holds.

\[\Box\]

### 4.4.4 Automation

In figure 4.1 we give two rules for symbolic execution serving as the proof required for a specification refinement of a method or function. Thus, we outline how to automatize the verification (i.e find the require proof) of $T \vdash \{A\} \{A'\} \Rightarrow \{B\} \{B'\}$.

#### Justifying the Automation for Methods

Concerning the precondition, the general idea is to show that the precondition $A_{pre}$ can be derived from $B_{pre}$. The consumption of the dept-free expression is necessary to ensure that $A_{pre}$ does not require more credits than required/provided by $B_{pre}$.

The production of $A_{post}$ will simulate the effect of the execution of the unknown body analogously to a method call or function application. This is the reason why we define the old-heap to be $\sigma_1.h$ and then produce into $\sigma_2$. After the production of $A_{post}$, $\sigma_3.h$ contains on the one hand the heap chunks given by $B_{pre}$ but not involved in $A_{pre}$, and on the other hand the heap chunks returned by $A_{post}$.

As a side remark, producing $A_{post}$ into an empty heap would make us lose all heap chunks given by $B_{pre}$ but not used by $A_{pre}$. The resulting notion of behavioural subtyping would be the one corresponding to the standard pairs of implication, that is weakening of precondition and strengthening of postcondition.

Having simulated the effect of the unknown body over which the specification $(A_{pre}, A_{post})$ abstracts, we consume $B_{post}$ and check again that we do not produce negative credits thereby.

#### Justifying the Automation for Functions

The rule for function deviates from the rule for methods as functions are side-effect free. Thus, we can drop the consumption of the dept-free expressions and we can produce $A_{post}$ into the same heap, namely $\sigma_1$, as we consumed the $A_{pre}$ from.

### 4.4.5 Incompleteness Issue

Unfortunately, the symbolic execution rule presented has an incompleteness problem arising from abstract predicate families. As long as the consume-rule is not
assertSpecificationRefinementOfMethod(a, b) =
  let
  fresh₁ = fresh, fresh₂ = fresh, ..., freshₙ = fresh
  fresh_{n+1} = fresh, ..., fresh_{n+m} = fresh
  in
  let
    γ = {(this, fresh),
          (x¹ᵃ, fresh₁), ..., (xⁿᵃ, freshₙ),
          (x¹ᵇ, fresh₁), ..., (xⁿᵇ, freshₙ),
          (y¹ᵃ, fresh_{n+1}), ..., (yᵐᵃ, fresh_{n+m}),
          (y¹ᵇ, fresh_{n+1}), ..., (yᵐᵇ, fresh_{n+m})}
  in
  produce(γ, ∅, ∅, {this ≠ null}, B_{pre}, (λ σ₁ •
    consume(σ₁, full, A_{pre}, (λ σ₂ •
      consume(σ₂, full, DeptFreeExpr, (λ _ • true))
      ∧
      produce(σ₂[σ₁.h / g], fresh, full, A_{post}, (λ σ₃ •
        consume(σ₃, full, B_{post}, (λ σ₄ •
          consume(σ₄, full, DeptFreeExpr, (λ _ • true)))))))})

where a and b are methods, x¹ᵃ, ..., xⁿᵃ respectively x¹ᵇ, ..., xⁿᵇ their parameters, y¹ᵃ, ..., yᵐᵃ respectively y¹ᵇ, ..., yᵐᵇ their return values and (A_{pre}, A_{post}) respectively (B_{pre}, B_{post}) their specifications for which the specification refinement shall be verified.

assertSpecificationRefinementOfFunction(a, b) =
  let
    fresh₁ = fresh, fresh₂ = fresh, ..., freshₙ = fresh
    fresh_y = fresh
  in
  let
    γ = {(this, fresh),
          (x¹ᵃ, fresh₁), ..., (xⁿᵃ, freshₙ),
          (x¹ᵇ, fresh₁), ..., (xⁿᵇ, freshₙ),
          (yᵃ, fresh_y), (yᵇ, fresh_y)}
  in
  produce(γ, ∅, ∅, {this ≠ null}, B_{pre}, (λ σ₁ •
    consume(σ₁, full, A_{pre}, (λ _ • true))
    ∧
    produce(σ₁, fresh, full, A_{post}, (λ σ₂ •
      consume(σ₂, full, B_{post}, (λ _ • true))))))

where a and b are functions, x¹ᵃ, ..., xⁿᵃ respectively x¹ᵇ, ..., xⁿᵇ their parameters, yᵃ respectively yᵇ their return value and (A_{pre}, A_{post}) respectively (B_{pre}, B_{post}) their specifications for which the specification refinement shall be verified.

Figure 4.1: Automation for the verification of specification refinements. Successful execution of the rule implies that the specification of method/function b refines the specification of method/function a.
able to automatically exchange abstract and concrete view of abstract predicates, consumption fails if a change in view is necessary. This will typically be necessary whenever a method or function gets inherited. We discuss this in more detail.

Abstract predicates abstract over access permissions and may thereby abstract over other abstract predicates. For the body verification, the programmer is required to explicitly make access permissions contained in abstract predicates available when they are required. The means for that are the ghost statements and the ghost expression. When it comes to the verification of specification refinements, the programmer should not be forced to write ghost statements or ghost expressions. As a consequence, it is the verifier’s task to exchange the concrete view and abstract view of abstract predicates in order to find the required access permissions.

As access permissions are required when consuming, we denote the algorithm for searching the access permissions the consume heuristics. Searching the access permissions seems to be hard as there may be many abstract predicates and as abstract predicates may be wrapped over multiple levels. Even worse, there may be recursive predicates and the question is when to stop unfolding/folding. Furthermore, with fractional permissions, there may even arise ambiguity. Several independent abstract predicates may contain a certain read permission. However, how do we know which abstract predicate to unfold in order to consume the read permission? In such cases, the verification may fail at a later point due to having unfolded the wrong abstract predicate.

As we did not see a stringent reason arguing for the support of a search heuristics at the time of this design decisions, we decided against a consume heuristics. This forces us to deal explicitly with the views of abstract predicates when verifying specification refinements. However, the resulting rules will be very similar and only involve additional steps for changing between abstract and concrete view of abstract predicates.

As a side remark: If the consume-rule were to use a heuristics for exchanging abstract and concrete views as proposed in the outlook section 7.2.1, the rules presented 4.1 could be reconsidered for an implementation.

### 4.5 Supporting Subclassing

In the following, we will discuss support for subtyping and inheritance separately. We will give the necessary proof obligations for both aspects. The automation of the proof obligations will be discussed in chapter 5 when discussing the implementation.

The proof obligations for support of inheritance are concerned with ensuring that inheriting members is sound. For that it ensures a specification refinement between the implementation specification and the interface specification of all classes as shown in figure 4.2.

The proof obligations for support of behavioural subtyping are concerned with enforcing behavioural subtyping. For that they ensure a specification refinement between the interface specification and the type specification of all classes as well
4.6 Supporting Subtyping

In this section we will outline the new proof obligations needed to support subtype polymorphism.

The proof obligations are:

- Behavioural subtyping, which is needed for supertype abstraction to be sound.
- Type specification conformance, which is needed to ensure that an implementation conforms to a specific type specification.

Furthermore, we will have to adapt the body verification in order to deal with dynamic dispatch.
We will present the proof obligations in the above order.

### 4.6.1 Behavioural Subtyping

The *behavioural subtyping proof obligation* is needed in order to guarantee that supertype abstraction used for the verification of dynamically dispatched member invocations is sound. The presented proof obligation works on the type specifications, thus not requiring to know any implementation. This enables modular verification of behavioural subtyping as the type specifications are available across module boundaries.

**Proof rule 4.4** Proof rule for establishing that a member ‘d’ is a behavioural subtype of a member ‘c’

Let \((C, C')\) and \((D, D')\) be the type specifications of \(c\) and \(d\), respectively. The proof rule is satisfied if \((C, C')\) refines \((D, D')\). Formally,

\[ T \vdash \{D\} \rightarrow \{D'\} \Rightarrow \{C\} \rightarrow \{C'\} \]

The proof rule can be interpreted as \((D, D')\) being the specification of the dynamically bound implementation and \((C, C')\) being the specification of the static type used for supertype abstraction.

We state the proof obligation required to enforce behavioural subtyping.

**Proof obligation 4.5** Let \(P\) be the program to verify. For all classes of \(P\) that specify a superclass, the behavioural subtyping proof rule 4.4 has to hold for all methods and functions.

### 4.6.2 Type Specification Conformance

The *type specification conformance proof obligation* is needed in order to guarantee that the type specification of a member gets satisfied by the member’s implementation. Instead of re-verifying the bodies against the type specification, we use the interface specification of the members to abstract over the body. Thus the proof rule does not depend on the body of the member.

**Proof rule 4.6** Type conformance of a member \(m\) of class \(C\)

Let the interface specification of \(m\) be \((I, I')\) and the type specification of \(m\) be \((T, T')\). The proof rule is satisfied if the type specification refines the interface specification of the member \(m\) given the additional assumption that the dynamic class of \(this\) is \(C\). Formally,

\[ T \vdash \{I\} \rightarrow \{I'\} \Rightarrow \{T \land \text{classOf(this)} = \text{C}\} \rightarrow \{T'\} \]

The additional assumption that the dynamic class of \(this\) equals \(C\) is motivated by the need to have an interpretation for *dynamic* symbol which appears as an index in the abstract families of the specifications. Recalling that the type specification explicitly uses the *dynamic* symbol in order to abstract over the concrete indices given in the interface specification, the additional assumption allows us to check
that the dynamic is used consistently in the type specification. However, the additional assumption restricts us to use the specification refinement only if we know the dynamic class of this.

**Proof obligation 4.7** Let $P$ be the program to verify. All methods and functions of all classes of $P$ have to satisfy the type conformance proof rule 4.6.

### 4.6.3 Supertype Abstraction

We presented the idea of supertype abstraction in section 2.3.2. In this section we give a definition of our exact understanding of supertype abstraction and give a proof sketch that it is sound to use the type specification of the static type’s executed member instead of the interface specification of the dynamically bound implementation.

**Definition 4.8** Let $\text{rcvr.memb(pars)}$ be a dynamically dispatched method call or function application. When using the type specification of the $\text{rcvr}$’s static type as abstraction for the statically unknown interface specification of the executed member, we call this supertype abstraction.

Note that we use the interface specification and not the implementation specification of the dynamically bound member. This is exactly in order to abstract over how the implementation was realized, and because only the interface specification but not the implementation specification is available across module boundaries.

Figure 4.4 gives an overview over the involved specification refinements.

![Figure 4.4](image-url)

**Figure 4.4:** A simple class hierarchy and the specification refinements involved. The solid arrows indicate a specification refinement. The thin arrows indicate generation of a specification. The implementation and interface specification of the classes C and X have been omitted due to space constraints. However, the situation for C and X is analogously to class D for which the specifications are given. We have included the implementation specification for completeness.
**Theorem 4.9** Supertype abstraction is sound

**Proof sketch** Let \( \text{rcvr}.\text{memb}(\text{pars}) \) be a dynamically dispatched method or function call. According to the definition of supertype abstraction, the specification used to verify the call is the type specification \( T_{\text{memb}} \) of the \( \text{rcvr} \)'s static type \( T \).

Assume without loss of generality that the dynamic class of \( \text{rcvr} \) is some class \( D \). Let \( DT \) by the type specification of \( D \) and let \( DT_{\text{memb}} \) be the type specification of \( \text{memb} \). Dynamic dispatch will execute the member \( \text{memb} \) in class \( D \) with interface specification \( DI_{\text{memb}} \).

Given the proof obligation for behavioural subtyping and type conformance, we show that the type specification of the receivers static type is a sound abstraction of the unknown static specification of the executed body. That is, we show that \( (T_{\text{memb}}, T'_{\text{memb}}) \) refines \( (DI_{\text{memb}}, DI'_{\text{memb}}) \).

The type conformance proof obligation gives us

\[
T \vdash \{DI_{\text{memb}}\}, \{DI'_{\text{memb}}\} \Rightarrow \{DT_{\text{memb}} \land \text{classOf}(\text{this}) = D\}, \{DT'_{\text{memb}}\}
\]

The behavioural subtyping proof obligation gives us

\[
T \vdash \{DT_{\text{memb}}\}, \{DT'_{\text{memb}}\} \Rightarrow \{T_{\text{memb}}\}, \{T'_{\text{memb}}\}
\]

As the behavioural subtyping rule guarantees the existence of a proof for all dynamic classes of \( \text{this} \), the rule holds as well for a particular dynamic class.

\[
T \vdash \{DT_{\text{memb}} \land \text{classOf}(\text{this}) = D\}, \{DT'_{\text{memb}}\} \Rightarrow \{T_{\text{memb}} \land \text{classOf}(\text{this}) = D\}, \{T'_{\text{memb}}\}
\]

Observe that neither the type conformance proof obligation nor the behavioural subtyping proof obligation depends on the body verification, which is needed to avoid a circular argument.

Combining the two yields.

\[
T \vdash \{DI_{\text{memb}}\}, \{DI'_{\text{memb}}\} \Rightarrow \{T_{\text{meth}} \land \text{classOf}(\text{this}) = D\}, \{T'_{\text{memb}}\}
\]

\[\square\]

### 4.7 Supporting Inheritance

When inheriting a member from a superclass, it has to be shown that the interface specification of the inheriting class refines the inherited member’s implementation specification.

A straightforward approach is to re-verify the inherited member’s implementation against the interface specification of the inheriting class. However, due to module boundaries the implementation may not always be available. Therefore, the presented formalism follows a different approach only using verification specifications.
4.7.1 Interface Specification Conformance

The interface specification conformance proof obligation is needed in order to guarantee that the interface specification of a member gets satisfied by the member’s implementation. Instead of re-verifying the bodies against the interface specification, we use the implementation specification of the members to abstract over the body. Thus the proof rule does not depend on the body of the member. Observe, that we thereby check that inheriting a particular member from a superclass is sound.

Proof rule 4.10 Interface conformance of a member \( m \) of class \( C \)

Let the implementation specification of \( m \) be \((B, B')\) and the interface specification be \((I, I')\). The proof rule is satisfied if the interface specification refines the implementation specification. Formally,

\[
T \vdash (B, B') \Rightarrow (I, I')
\]

Proof obligation 4.11 Let \( P \) be the program to verify. All methods, functions and constructors of all classes of \( P \) have to satisfy the interface conformance proof rule 4.10.

Observe that the proof rule is syntactically equivalent to the behavioural subtyping proof rule 4.4. It only differs in the kind of the used specifications and the fact that the specifications stem from different classes.

4.8 Adapting the Symbolic Execution Rules

We detail in this section how existing symbolic execution rules had to be adapted in order to support the new language features.

4.8.1 Dynamic Class

We often will have to know the dynamic class of an instance. With dynamic class we mean the effective class the object has been instantiated from. For that reason we introduce a function symbol \( \text{dynamicClassOf} \) and the sort \( \text{Class} \). In the following, we will represent terms of the sort \( \text{Class} \) directly by the class name - usually a capital letter \( C \) or \( D \).

The function symbol \( \text{dynamicClassOf} \) is given by

\[
\text{dynamicClassOf} : \text{Ref} \rightarrow \text{Class}
\]

Note that we refer to the class and not to the type to emphasize that we are indeed interested in the implementation and not the type.

4.8.2 Abstract Predicate Families

As we have introduced abstract predicate families, we have to adapt the symbolic representation of predicate chunks as well as the rules for consumption and
converse(σ, h, ta, acc(e.V[τ], a), Q) =
  eval(σ, e, (λ σ1, t •
  eval(σ1, a, (λ σ2, ta0 •
  let ta2 = ta * ta0 in
  σ2 ⊢ t ≠ null
  ∧ (∃ pc @ t.V[τ'][tv] # ta1 ∈ h s.t
  j' = j
  ∨ (j = dynamic ∧ j' ≠ dynamic ∧ σ2 ⊢ dynamicClassOf(t) = j'))
  ∧ σ2 ⊢ ta2 > 0 ∧ ta2 ≤ ta1)
  if σ2 ⊢ ta2 = ta1
  Q(h - pc, σ2.π, tv)
  else
  Q(h - pc + t.V[τ'][tv] # ta1 - ta2, σ2.π, tv)))))

produce(σ, tv, ta, acc(e.V[τ], a), Q) =
  eval(σ, e, (λ σ1, t •
  eval(σ1, a, (λ σ2, ta1 •
  σ2 ⊢ t ≠ null ∧ ta1 > 0 ∧
  let h3, π3 = merge(σ2.h, {t.V[τ] # (ta * ta1)}) in
  Q(h3, σ2.π + π3))))

  eval'(σ, π, rs, unfolding acc(e0.V[τ], a) in e1, Q) =
  eval'(σ, π, rs, e0, (λ π1, t0 •
  eval'(σ, π1, rs, a, (λ σ2, ta •
  σ + σ2 ⊢ t0 ≠ null ∧ ta ≥ 0 ∧
  consume(σ + σ2, full, acc(e0.V[τ], a), (λ σ1, tv •
  produce(σ1[(this, t0) / γ], tv, ta, body1.V, (λ σ2 •
  eval'(σ2[σ:̸γ / g, σ.γ / γ, ⊥ / π], σ2.π, rs, e1, Q))))))))

exec(σ, fold acc(e.V[τ], a), Q) =
  eval(σ, e, (λ σ1, t •
  eval(σ1, a, (λ σ2, ta •
  σ2 ⊢ t ≠ null ∧ ta ≥ 0 ∧
  consume(σ2 / (this, t), ta, body1.V, (λ σ3, tv •
  let (hr, πr) = merge(σ3.h, {t.V<tv> # ta} in
  Q(σ3 / hr + πr))))))))

exec(σ, unfold acc(e.V[τ], a), Q) =
  eval(σ, e, (λ σ1, t •
  eval(σ1, a, (λ σ2, ta •
  σ2 ⊢ t ≠ null ∧ ta ≥ 0 ∧
  consume(σ2, full, acc(e.V[τ], a), (λ σ3, tv •
  produce(σ3 / (this, t), tv, ta, body1.V, (λ σ4 •
  Q(σ4 / σ2.γ))))))))

Figure 4.5: Symbolic execution rules affected by the introduction of abstract predicate families. In comparison to the symbolic execution rules given by Schwerhoff, we introduced and took care of the abstract predicate family index. V is an abstract predicate and body1.V is the body of predicate V in class I. J is either the special symbol dynamic or a concrete class, while I is always a concrete class.
4.8. ADAPTING THE SYMBOLIC EXECUTION RULES

production of abstract predicates. Furthermore, we have to adapt the symbolic execution of _unfold_, _fold_ and _unfolding_.

We give the new definition for predicate chunks where we have only added the abstract predicate family index with respect to the definition given by Schwerhoff.

**Definition 4.12** Predicate chunks are of the form \( t_r.V^{\{J\}[t_s]} # t_a \), where

- \( t_r \) represents a receiver object
- \( V \) represents the predicate family of the receiver object
- \( J \) represents the abstract predicate family index. It is a concrete class or the special symbol _dynamic_
- \( t_s \) is a snapshot determining the values of the fields that the predicate \( t_r.V^{\{J\}} \) abstracts over, i.e. the heap segments covered by \( t_r.V^{\{J\}} \)
- \( t_a \) is a fractional permission representing the current thread’s access to the predicate

Figure 4.5 gives the symbolic execution rules that were adapted as a consequence of the introduction of abstract predicate families.

### 4.8.3 Dynamically Dispatched Member Invocations

Subtype polymorphism affects body verification by the presence of dynamically dispatched calls and function applications. Furthermore, we have introduced the verification specifications and therefore need to clarify when which specification gets used during body verification. As a consequence, we restate the symbolic execution rules for calls and function applications as well as all rules depending on specifications.

**Function Applications**

In figure 4.6 we state the evaluation rules for statically- and dynamically bound function applications replacing the rule for evaluating a function application as given by Schwerhoff. In order to distinguish statically- and dynamically bound function applications we use the syntax \( \text{rcvr}::p \) for a statically bound function application and \( \text{rcvr}.p \) for a dynamically bound function application. We as well have to adapt the function symbols used for symbolic execution.

For each pure \( n \)-ary Chalice function

\[
p(x_1 : T_1, \ldots, x_n : T_n) : T_r
\]

introduced, inherited or respecified in a class \( C \) there exists a function symbol

\[
C:: p : \text{Snap} \times \text{Ref} \times S_1 \times \ldots \times S_n \rightarrow S_r
\]
Figure 4.6: Symbolic evaluation of statically- and dynamically bound function applications. In order to
 distinguish them, we use \( c :: p \) for statically- and \( c.p \) for dynamically bound function applications. \( \text{cons}(a, \text{L}) \) prepends the element \( a \) to the list \( \text{L} \).
exec(σ, call r1, ..., rm := e:: m(ex1, ..., exn), Q) =
eval(σ, e, (λ σ1, t •
σ1 ⊢ t ≠ null ∧
evals(σ1, [ex1, ..., exn], (λ σ2, [tx1, ...,txn] •
let
xγ = {(x1, tx1), ..., (xn,txn)},
yγ = {(y1, fresh), ..., (ym, fresh)}
in
consume(σ2 / xγ + (this, t), full, iprecCm, (λ σ3, _ •
produce(σ3[σ2.h / g] + yγ, fresh, full, ipostCm & lkchCm, (λ σ4 •
Q(σ4[σ2.γ / γ , σ2.g / g] + {(r1, σ4.γ(y1)), ..., (rm, σ4.γ(ym))})))))

where C is the static type of the receiver e, (iprecCm, ipostCm) and lkchCm are the interface specification and lockchange clause, respectively, of the method to be invoked.

exec(σ, call r1, ..., rm := e.m(ex1, ..., exn), Q) =
eval(σ, e, (λ σ1, t •
σ1 ⊢ t ≠ null ∧
evals(σ1, [ex1, ..., exn], (λ σ2, [tx1, ...,txn] •
let
xγ = {(x1, tx1), ..., (xn,txn)},
yγ = {(y1, fresh), ..., (ym, fresh)}
in
consume(σ2 / xγ + (this, t), full, tprecCm, (λ σ3, _ •
produce(σ3[σ2.h / g] + yγ, fresh, full, tpostCm & lkchCm, (λ σ4 •
Q(σ4[σ2.γ / γ , σ2.g / g] + {(r1, σ4.γ(y1)), ..., (rm, σ4.γ(ym))})))))

where C is the static type of the receiver e, (tprecCm, tpostCm) and lkchCm are the type specification and lockchange clause, respectively, of the receiver’s static type.

Figure 4.7: Symbolic execution of dynamically- and statically bound method calls. In order to distinguish them, we use e:: m for statically- and e.m for dynamically bound method calls.

and in addition, for each n-ary Chalice function introduced (not overridden, re-specified or inherited) in a class C there exists a function symbol

C.p : Snap × Ref × S1 × ... × Sn → Sf

in T's signature, where Si is a sort corresponding to Chalice’s type Ti.

Method Calls

In figure 4.7 we state the execution rules for statically- and dynamically bound method calls replacing the rule for executing a call as given by Schwerhoff. We use the same syntax as for function applications in order to distinguish between statically- calls and dynamically bound method calls. The rules we give differ from the rules from Schwerhoff in the specifications involved.
4.8.4 Fork and Join

In noSubclassing-Chalice, all calls were bound statically. In subclassing-Chalice, calls can be bound statically or dynamically. This applies as well for asynchronous method calls. The symbolic execution rules for the fork and join statement have to be adapted in an analogous way as we did for synchronous method calls. That is, we have to use the appropriate specifications depending on whether the asynchronous call is statically or dynamically bound.

The symbolic execution rule for the join statement depends on a token chunk, generated by the symbolic execution of the fork statement, in order to produce the postcondition of the asynchronously called method. In subclassing-Chalice, the postcondition to use depends in addition on how the call was bound. If a call was statically bound we have to use the interface specification and if the call was dynamically bound we have to use the type specification of the receiver’s static type.

We therefore have to adapt the representation of the token chunk. We differ in two points from the definition given by Schwerhoff. Firstly, we differ in the way we identify the invoked method. Independently of whether the invocation was statically or dynamically bound, we use the symbol $C_m$ to express that the method $m$ of class $C$ was executed from a static perspective. That is, the receiver’s static type used in the fork statement was $C$.

Secondly, we introduce an additional boolean flag indicating whether the asynchronous call was statically or dynamically bound. We state the adapted definition based on the one from Schwerhoff, whereby we differ only in the mentioned points.

**Definition 4.13** Token chunks are of the form $t_r \mapsto (C_m, db, h, t, ts)$, 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. Thereby, $C$ corresponds to the static type of the receiver of the asynchronous method call.
- $db$ is a boolean flag. The flag $db$ equals true if and only if the asynchronous call was dynamically bound.
- $h$ 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
- $ts$ represents the actual method arguments

The symbolic execution rules for the fork statement are given in figure 4.8 and the rule for the join statement is given in figure 4.9.

4.8.5 Constructors

Constructors are treated as special instances of methods for the verification of the body. That is, a constructor is valid if it satisfied the method validity rule given
exec(σ, fork ek := e:: m(ex1, ..., exn), Q) =
    eval(σ, e, (λ o1, t •
        eval(σ1, ek, (λ o2, tk •
            o2 ⊢ t ≠ null ∧
                evals(σ2, [ex1, ..., exn], (λ o3, [tx1, ...,txn] •
                    let xγ = {(x1, tx1), ..., (xn,txn)} in
                    consume(σ3 / xγ + (this, t), full, ipreCm, (λ o4, _ •
                        Q(σ4[σ3.γ / γ, o3.g / g]
                            - tk.joinable ⇒ # _
                            + tk.joinable ⇒ true # eps
                            - tk ⇒ (ω ω ω _)
                            + tk ⇒ (Cm, false, o2.h, t, [tx1, ...,txn]))))
                    )
                )
            )
        )
    )
where C is the static type of the receiver e and (ipreCm, ipostCm) is the interface specification of the method to be invoked asynchronously.

exec(σ, fork ek := e.m(ex1, ..., exn), Q) =
    eval(σ, e, (λ o1, t •
        eval(σ1, ek, (λ o2, tk •
            o2 ⊢ t ≠ null ∧
                evals(σ2, [ex1, ..., exn], (λ o3, [tx1, ...,txn] •
                    let xγ = {(x1, tx1), ..., (xn,txn)} in
                    consume(σ3 / xγ + (this, t), full, tpreCm, (λ o4, _ •
                        Q(σ4[σ3.γ / γ, o3.g / g]
                            - tk.joinable ⇒ # _
                            + tk.joinable ⇒ true # eps
                            - tk ⇒ (ω ω ω _)
                            + tk ⇒ (Cm, true, o2.h, t, [tx1, ...,txn]))))
                    )
                )
        )
    )
where C is the static type of the receiver e and (tpreCm, tpostCm) is the type specification of the method to be invoked asynchronously.

Figure 4.8: Symbolic execution of dynamically- and statically bound asynchronous method calls. In order to distinguish them, we use e:: m for statically- and e.m for dynamically bound asynchronous method calls. The two presented rules differ from the one by Schwerhoff in the specifications used and the arguments to the token chunk.
exec(σ, join r1, ..., rm := ek, Q) =
eval(σ, ek, (λ σ1, tk •
σ1 ⊢ tk ≠ null
∧ ∃ fc @ tk.joinable ⊢ true # ta ∈ σ1.h
∧ ∃ tc @ tk ⊢ (Cm, db, g1, t, [tx1, ..., txn]) ∈ σ1.h ∧
let
xγ = {(x1, tx1), ..., (xn, txn)},
yγ = {(y1, fresh), ..., (ym, fresh)},
σ2 = σ1[g1 / g] / xγ + yγ + (this, t)
post = if db = true
   tpostCm && lkchCm
else
   ipostCm && lkchCm
in
produce(σ2, fresh, full, post, (λ σ3 •
evals(σ3, lkchCm, (λ σ4, [t1, ..., tl] •
Q(σ4[σ1.g / g, σ1.γ / γ]
   − fc − tc
   + {(r1, σ4.γ(y1)), ..., (rm, σ4.γ(ym))}
   + tk.joinable ⊢ false # ta)))
)

where (tpreCm, tpostCm) is the type specification, (ipreCm, ipostCm) the interface specification and lkchCm the lockchange expression of the method m in class C.

Figure 4.9: Symbolic execution for the join statement. The rule differs from the one by Schwerhoff in the specification used.

by Schwerhoff. Constructor calls from within other constructors are treated as method calls and the symbolic execution rule for method calls applies.

However, we need to provide the symbolic execution rule for the instantiation of new objects and how to execute the special \texttt{InitDeclaredFields} statement being introduced implicitly after the super constructor call in a primary constructor.

The rule for the symbolic execution of an instantiation as well as the rule for executing \texttt{InitDeclaredFields} is given in figure 4.10.

4.8.6 Object Sharing

With the new semantics of monitor invariant presented in section 3.5.12 the symbolic execution of the \texttt{share}, \texttt{acquire}, \texttt{release} and \texttt{unshare} has to be adapted. First of all, we have to ensure that sharing an object is only possible if the dynamic class equals the static type of the involved instance. Secondly, we have to ensure that an object gets released using the same static type as it has been acquired with.

That the object is released at the same static type as it was acquire is ensured by adapting the lock modes. The read lock mode, denoted ‘R’, and the write lock mode, denoted ‘W’, are associated with the static type required for releasing.
4.8. ADAPTING THE SYMBOLIC EXECUTION RULES

exec(σ, x := new C(ex1, ..., exn), Q) =
evals(σ, [ex1, ..., exn], (λ σ2, [tx1, ...,txn] •
let
  xγ = {(x1, tx1), ..., (xn,txn)},
  t = fresh, tµ = bottomlock
in
let
  σ3 = σ2 + (t ≠ null) + dynamicClassOf(t) = C
  + t.μu → tμ # full + mu(t, σ.μ(vm))) = tµ
in
  consume(σ3 / xγ + (this, t), full, ipre, (λ σ4, •
    produce(σ4[σ3.h / g], fresh, full, ipost & lkch, (λ σ5 •
      Q(σ5[σ3.γ / γ, σ3.g / g] + (x, t)))))))
where (ipre, ipost) is the interface specification of the constructor.

exec(σ, InitDeclaredFields, Q) =
eval(σ, this, (λ σ2, t •
  Q(σ2 + {t.f1 → fresh # full, ..., t.fn → fresh # full})
where f1 to fn are all declared fields (i.e. not inherited ones) of class C.

---

Figure 4.10: Symbolic execution of an object instantiation and the special InitDeclaredFields statement.

One important property of our implemented semantics for monitor invariants is that the defined monitor invariants have to be self-framing. However, as this is not a new requirement and was already required by Schwerhoff we do not restate the corresponding symbolic execution rule or proof obligation but only emphasize its existence.

Figures 4.11 and 4.12 give the symbolic execution rules.
exec(σ, share e between el and eu, Q) =

eval(σ, e, (λ σ1, t •
σ1 ⊢ t ≠ null
∧ ∃ fc @ t.mu ⤸ lockbottom # ta ∈ σ1.h
∧ σ1 ⊢ ta ≥ 100
∧ σ1 ⊢ dynamicClassOf(t) = C ∧
  let
  pre = el1 << eu1 & & ... & & el1 << eu1 & & ... & & eln << eu1 & & ... & & eln << eu1,
  post = el1 << e & & ... & & eln << e & & e << eu1 & & ... & & e << eu1
  in
  consume(σ1, full, pre, (λ σ2, _ •
    let
    tµ = fresh,
    v = σ2.h(v) + 1,
    σ3 = σ2 / σ2.h[v++]
    + ∀ o ≠ t µ( t, v) = µ( t, v - 1)
    + µ( t, v) = tµ + (tµ ≠ lockbottom)
    + holds(t, σ4.h(v)) = N
    - fc + t.mu ⤸ tµ # ta
  in
  eval(σ3, post, (λ σ4, tw •
    consume(σ4 / (this, t) + tw, full, invC, (λ σ5, _ •
      Q(σ5 / σ4.(γ)))))))))))
where the upper bounds eu contain neither waitlevel nor lockbottom, where C denotes the static type of reference e and where invC is the effective monitor invariant of the object's class.

exec(σ, unshare e, Q) =

eval(σ, e, (λ σ1, t •
σ1 ⊢ t ≠ null
∧ (holds(t, σ1.h(v)) = W(C))
∧ ∃ fc @ t.mu ⤸ # ta ∈ σ1.h
∧ σ1 ⊢ ta ≥ 100 ∧
  let v = σ1.h(v) + 1 in
  Q(σ1 / σ1.h[v++]
  - fc + t.mu ⤸ lockbottom # ta
  + µ(t, v) = lockbottom
  + ∀ o ≠ t µ(t, v) = µ(t, v - 1)
  + (holds(t, v) = N)
  + (∀ o ≠ t holds(t, v) = holds(t, v - 1)))))))
where C is the static class of e.

---

Figure 4.11: Symbolic execution of share and unshare statements.
exec(σ, acquire(e, m), Q) =
eval(σ, e, (λ σ1, t •
    let
      tm = if (m = W) W(C) else R(C)
      ta = if (m = W) full else eps
    in
      σ2 ⊢ t ≠ null ∧
        consume(σ2, full, waitlevel << e.mu, (λ σ3, _ •
          let
            v = σ3.h(v_h) + 1,
            σ4 = σ3 / (this, t) / σ3.h[v_h++]
            + (holds(t, v) = tm)
            + (∀o ≠ t • holds(t, v) = holds(t, v − 1))
          in
            produce(σ4, fresh, ta, inv_C, (λ σ5 •
              Q(σ5 / σ2.γ))))))
    where C is the static type of e and where inv_C is the effective monitor invariant of C.

exec(σ, release(e, m), Q) =
  eval(σ, e, (λ σ1, t •
    let
      tm = if (m = W) W(C) else R(C)
      ta = if (m = W) full else eps
    in
      σ2 ⊢ t ≠ null ∧ (holds(t, σ2.h(v_h)) = tm) ∧
        consume(σ2 / (this, t), ta, inv_C, (λ σ3, _ •
          let v = σ3.h(v_h) + 1 in
            Q(σ3 / σ2.γ / σ3.h[v_h++]
              + (holds(t, v) = N)
              + (∀o ≠ t • holds(t, v) = holds(t, v − 1))))))
  where C is the static type of e and where inv_C is the effective monitor invariant of C.

Figure 4.12: Symbolic execution of acquire and release statements.
The goal of this chapter is to detail how the verification support for subtype polymorphism and inheritance has been realized in Syxc. First we present the software architectural changes and how the representation of the symbolic heap had to be changed. Following that, we explain how the proof obligations for behavioural subtyping, type conformance and interface conformance were automatized.

In the following we will describe our changes relative to the presentation of Syxc by Schwerhoff [24]. We will assume that the reader is familiar with the general architecture and the terms *Evaluator, Executor, Consumer, Producer, Decider* and *Prover*.

### 5.1 General Remarks on The Implementation

The implementation resulting from this work is split over two code bases. On the one hand side we have adapted the implementation of the Chalice program verifier and on the other hand the program verifier Syxc. The reason is that Syxc depends on the Chalice parser and type-checker.

When adapting Chalice, great care has been taken not to break backward compatibility with older Chalice programs. The reason is that we have only adapted the parser and type-checker of Chalice and not the program verifier part of it. That is, while the parser and type-checker of Chalice can deal with subclassing-Chalice programs, the program verifier can not verify them.

As a general implementation decision, all code depending on the Chalice-AST is contained in the Chalice code base, even if the code is not used there. The reason is that it may be required when the Chalice program verifier gets extended with support for the language features added for subclassing-Chalice. All code working on the Syxc-AST is for obvious reasons contained in the Syxc code base.

We provide tests for Chalice and Syxc. The tests for Chalice ensures that subclassing-Chalice programs get parsed correctly and that the language restrictions of subclassing-Chalice are enforced properly. The fine-grained tests for Syxc ensure that subclassing-Chalice programs get correctly verified.
5.2 Software-Architectural Changes in Syxc

In the version of Syxc prior to this work, body verification of all kinds of members was the main concern. With the addition of support for subclassing, new aspects have to be verified. In particular behavioural subtyping and interface conformance has to be checked. In order to separate the verification of the different aspects we introduce the concept of *aspect verifiers*. For each of the aspects body verification, attesting behavioural subtyping and interface conformance we introduce a dedicated aspect verifier. It will become clear in the following why we do not have an aspect verifier for the type conformance proof obligation.

The aspect verifiers are orchestrated by the *program verifier* whose task is to coordinate the aspect verifiers and to support parallelism. In order to support parallelism in the verification, the concept of *verification tasks* has been introduced. As the first step of verifying a program, all verification tasks are generated. In a second step the verification tasks are executed using the aspect verifiers. While this conceptually allows parallelism, as a last step one would have to deal with concurrent instances of the Decider. As it was not our goal to add support for parallelism this last step has not been taken.

5.3 New Representation of Predicate Chunks

With the addition of abstract predicate families the representation of predicate chunks had to be adapted. Prior to our work, predicate and field chunks were both identified by the pair of a *receiver* term and an *id* (name of predicate or field) allowing to treat predicate and field chunks to some extent generically. With abstract predicate families this changes.

**Supporting Abstract Predicate Families**

In order to distinguish abstract predicate chunks representing abstract predicates of the same abstract predicate family, we need to add the abstract predicate family index to the predicate chunks. Thus, a predicate chunk gets identified by the triple \((rcvr, id, index)\) where \(rcvr\) is the receiver of the predicate access, \(id\) is the name of the abstract predicate and \(index\) the abstract predicate family index. As in our formalism, we allow a predicate chunk to have an index corresponding to the special *dynamic* symbol we introduced in definition 3.4. Thereby, we do not need to deal with abstract predicate families explicitly.

This change in representation has to be reflected in the Decider and Prover as well. We do not discuss these changes in the following.

**Identifying Field and Predicate Chunks Generically**

While predicate chunks are identified using a triple \((rcvr, id, index)\) field chunks still are identified by the pair \((rcvr, id)\). In order to abstract over the difference we
introduce the concept of ChunkIdentifier. More precisely, there are FieldChunkIdentifier and PredicateChunkIdentifier having a common base class ChunkIdentifier. With the help of chunk identifiers we do, for instance, not need to provide separate implementations for looking up a field chunk or predicate chunk in the heap.

5.4 Dealing with Abstract Predicates

In section 3.5.8 we presented the notion of inheriting a predicate by aggregation. We first recall the definition of inheriting a predicate by aggregation and then outline another notion that we could have implemented.

5.4.1 Inheriting a Predicate by Aggregation

The semantics of inheriting a predicate by aggregation is equivalent to overriding the predicate of the superclass and providing a predicate body containing only the superclass' predicate.

Let $D$ be the subclass of $C$ and let $v_{<C>} := b$ be the predicate $V$ defined in $C$ with body $b$. Using the notion of inheriting the predicate by aggregation, we have that $v_{<D>} := \text{this}.v_{<C>}$. Employing this notion, a programmer will normally - in order to align with the semantics of inheritance - override predicates such that the superclass' predicate is aggregated as well.

5.4.2 Inheriting Predicate Bodies

The semantics of inheriting the bodies of predicates is equivalent to overriding the predicate of the superclass and providing the same body as the superclass' predicate.

Let $D$ be the subclass of $C$ and let $v_{<C>} := b$ be the predicate $V$ defined in $C$ with body $b$. Using the notion of inheriting the predicate by aggregation, we have that $v_{<D>} := b$. Employing this notion, a programmer will normally - in order to align with the semantics of inheritance - repeat the entire body of the superclass' predicate when overriding it.

5.4.3 Comparing the Notions of Predicate Inheritance

The notions have a crucial impact on how the programmer needs to use ghost statements and ghost expressions.

We discuss the difference at the example of a super call. Assume the programmer has an abstract predicate $\text{this}.v_{<D>}$ of some class $D$ and needs, for the super call,
the abstract predicate \texttt{this}.$V$\texttt{<C>} where \(D\) is a indirect subclass of \(C\). We assume that \(V\) was inherited by \(D\) and all its superclasses up to \(C\).

Using the notion of inheriting by aggregation, the programmer will have to unfold a series of predicates. Starting with unfolding \texttt{this}.$V$\texttt{<D>} he will have to unfold for all of \(D\)'s superclasses, thereby going up the subclassing hierarchy step by step until the desired abstract predicate \texttt{this}.$V$\texttt{<C>} gets unfolded. In the end the programmer will have to fold each of the abstract predicates again in the reverse order.

Using the notion of inheriting bodies, the programmer will have to unfold \texttt{this}.$V$\texttt{<D>} and then fold \texttt{this}.$V$\texttt{<C>}. Thus, fewer fold and unfold statements are required for the given example.

We have decided against notion of body inheritance in favour of inheriting a predicate by aggregation for the following reasons. First of all, the resulting code duplication did not seem to be favourable. Furthermore, thinking of inheritance across module boundaries, it is questionable whether the body of a predicate defined in another module should be inherited literally. Providing a special solution for module boundaries forces the programmer to handle module boundaries explicitly which is not necessary with the other notion. In addition, as a predicate tends to include all definitions of overridden or inherited predicates literally, unfolding such a predicate will provide the Prover with the implementation knowledge of all superclasses at once instead of only the superclasses the programmer deals with. This will in general increase the size of the symbolic heap and the number of path conditions the verifier has to deal with. As a last point we remark that this notion certainly would require support for a ‘folding’ expressions (the counterpart to the unfolding expression), which is not hard to support though.

### 5.5 Automating the Proof Obligations

In this section we explain how the proof obligations type conformance, interface conformance and behavioural subtyping were automated in Syxc.

#### 5.5.1 Type Specification Conformance

The type specification conformance proof obligation 4.7 ensures that the type specification of a class \(C\) refines the interface specification of \(C\) given an instance of class \(C\). As the verification specifications get produced as described in section 3.5.10, the type conformance proof obligation is satisfied by construction. Thus, we do not need to verify it given the current design.

**Theorem 5.1** If the type specifications are generated as described in section 3.5.10, the type conformance proof obligation is satisfied by construction.

**Proof sketch** Let \(m\) be an arbitrary member of an arbitrary class \(C\). Let \(C_m := (C^\text{pre}_m, C^\text{post}_m)\) be the code specification of \(m\), let \(I_m := (I^\text{pre}_m, I^\text{post}_m)\) be the interface
specification of \( m \) and let \( T_m := (T^\text{pre}_m, T^\text{post}_m) \) be the type specification of \( m \). We have to show that \( \Gamma \vdash \{ I^\text{pre}_m \}_m \{ I^\text{post}_m \} \Rightarrow \{ T^\text{pre}_m \land \text{classOf} (\text{this}) = C \}_m \{ T^\text{post}_m \} \) holds.

From the interface specification transformation given in section 3.5.10 we know the following definitions

1. If \( m \) is an external member, then all unspecified indices are \textit{dynamic}.
   That is \( I_m := I(\text{C}_m, \text{dynamic}) \)

2. If \( m \) is an internal member, then use the \textit{syntactical indexing transformation}.
   That is \( I_m := SI(\text{C}_m, \text{C}) \)

Furthermore, from the type specification transformation given in section 3.5.10 we know the following definitions

\[ T_m := I(\text{C}_m, \text{dynamic}) \]

Using \( T^\text{pre}_m \land \text{classOf} (\text{this}) = C \) we can re-index all abstract families associated with \text{this} and indexed by \text{dynamic} with the new index \text{C}. We can do this obviously for the type specification and for less obvious reasons as well for the interface specification. The later can be argued as follows. As we use \( \{ I^\text{pre}_m \}_m \{ I^\text{post}_m \} \) inside the proof of \( \{ T^\text{pre}_m \land \text{classOf} (\text{this}) = C \}_m \{ T^\text{post}_m \} \) and as \( \{ I^\text{pre}_m \}_m \{ I^\text{post}_m \} \) holds for the body without knowing the dynamic class of \text{this}, then it holds as well for the dynamic class being \text{C}. Let \( T'_m \) and \( I'_m \) be the re-indexed versions of \( T_m \) and \( I_m \) respectively where the equal sign means syntactical equality.

For the re-indexed interface specification we have

1. If \( m \) is an external member we have
   \[ I'_m := SR(I(\text{C}_m, \text{dynamic}), \text{C}, \text{dynamic}) = SR(SI(\text{C}_m, \text{C}), \text{C}, \text{dynamic}) \]

2. If \( m \) is an internal member we have
   \[ I'_m := SR(SI(\text{C}_m, \text{C}), \text{C}, \text{dynamic}) \]

For the re-indexed type specification we have

\[ T'_m := SR(I(\text{C}_m, \text{dynamic}), \text{C}, \text{dynamic}) = SR(SI(\text{C}_m, \text{C}), \text{C}, \text{dynamic}) \]

It follows readily that the two specifications are syntactically equal. \( \square \)

Informally, the argument can be paraphrased as follows. Whenever a predicate family associated with \text{this} and indexed by \text{dynamic} appears in the type specification, then the interface specification requires at the same position a predicate family associated with \text{this} and indexed by either \text{dynamic} or by \text{C} which using definitions 3.4, 3.10 and \text{classOf} (\text{this}) = \text{C} yields the desired result.

### 5.5.2 Interface Specification Conformance

The interface specification conformance proof obligation 4.11 ensures that the interface specification of a class \( C \) refines the implementation specification of class \( C \). Thereby, the proof obligation subsumes the check that it is sound to inherit a member from a superclass.
We handle the interface specification conformance proof obligation for overridden, respecified, inherited members separately.

**Interface Specification Conformance of Overriding Members**

The interface specifications for Overriding members get produced as described in section 3.5.10.

**Theorem 5.2** If the verification specifications are generated as described in section 3.5.10, the interface conformance proof obligation is satisfied by construction for all overriding members of a class.

**Proof sketch** Let \( m \) be an arbitrary overriding member of an arbitrary class \( C \). Let \( I_m := (I^\text{pre}_m, I^\text{post}_m) \) be the interface specification of \( m \) and let \( B_m := (B^\text{pre}_m, B^\text{post}_m) \) be the implementation specification of \( m \). We have to show that \( T \vdash \{ B^\text{pre}_m \}, \{ B^\text{post}_m \} \Rightarrow \{ I^\text{pre}_m \}, \{ I^\text{post}_m \} \) holds for a overriding member.

Recall how the verification specifications are generated as described in section 3.5.10.

By observing that for overriding members the transformations give exactly the same definition, it follows that the interface specification and implementation specification for overriding members are syntactically equal. □

**Interface Specification Conformance of Respecifying Members**

Respecifying members should be treated the same way as inherited members. That is, by verifying that the interface specification (given by the respecifying member) refines the implementation specification (given by the respecified member). For historical reasons and time constraints we did not achieve this any more. In the current implementation, respecifying members are reverified. That is, we treat respecifying members in an analogous way as overriding members. However, as the programmer does not provide a body for respecifying members, we have to generate one and thereby have to take care of the ghost-statements needed.

There are two possibilities to generate the body. We can either generate a body containing a super member invocation with the corresponding ghost statements or we transform the respecified body by injecting additional ghost statements. For historical reasons we have implemented both approaches. The first one is in particular needed to deal properly with module boundaries for which we are not allowed to use the second approach.

Given the current implementation for respecifying members, we can consider it as syntactic sugar as it is equivalent to an overriding member for which we generate the new body based on the overridden body. As the generated body gets verified against the specification provided by the respecifying member (i.e the programmer) an error in the body generation would make the body verification fail. That is, the correctness of the body generation algorithm is not relevant for soundness but only for completeness of our verification approach. Given that and the fact
that we consider the implementation obsolete, we do not detail the generation of the bodies and refer the interested reader to the code.

**Interface Specification Conformance of Inherited Members**

In order to facilitate automation of the interface conformance proof obligation we do not explicitly use the implementation specification of inherited members. The reason is that a member could be inherited over many classes making it hard to automate due to the required ghost statements.

We first present how we derive a replacement for the implementation specification of an inherited member allowing us to assume that a member is inherited from the direct superclass. Afterwards we detail the automation of the proof obligation.

Let $D$ be a direct subclass of some class $C$ and let $D$ inherit a member $m$ from $C$ or an indirect superclass. Let $BD_m$ denote the implementation specification of $m$ in $D$ and let $IC_m$ be the interface specification of $m$ in $C$. Instead of using the real implementation specification of the inherited member $m$, we use the interface specification $IC_m$ as the implementation specification $BD_m$. That is, we define $BD_m := IC_m$ for an inherited member.

**Theorem 5.3** Let $D$ be a direct subclass of some class $C$. Let $D$ inherit a member $m$ from the direct superclass $C$ or an indirect superclass. Using the interface specification of $m$ given for class $C$ instead of the real implementation specification of $m$ in class $D$ in the interface conformance proof obligation is sound.

**Proof sketch** The correctness follows by an inductive argument on the inheritance depth of a member.

The base case is given by the class defining the member (inheritance depth 0). As we have shown before, the interface conformance proof obligation is satisfied by construction for non-inherited member. That is, the interface specification of $m$ in the defining class is a refinement of its implementation specification of $m$.

The inductive step is given by the following argument. We use the notation from the theorem and assume that the induction hypothesis holds for class $C$. By the induction hypothesis we know that the interface specification of $m$ given in $C$ is a refinement of the real implementation specification of $m$. If we show that the interface conformance proof rule is satisfied for $m$ in class $D$, then we have that the interface specification of $m$ in $D$ is a refinement of the interface specification of $m$ in $C$.

By the transitivity of these refinements we conclude that the interface specification of $m$ in $D$ is a refinement of the real implementation specification of $m$. □

The interface proof rule is checked by our verifier for inherited members. We detail how we automated it in figure 5.1.

The main aspect we have to deal with is changing between the abstract and concrete view of abstract predicates appearing in the interface specification. We illustrate the issue with an example outlining that we have to unfold `this.valid<Recell>`
assertInterfaceConformanceOfInheritedMethod(Sub_m, Sup_m) =
let
  γ = {(this, fresh),
       (x_1, fresh), ..., (x_n, fresh),
       (y_1, fresh), ..., (y_m, fresh)
  }
in
produce(γ, ∅, ∅, {this ≠ null}, ipre_{Sub_m}, (λ σ1 •
unfoldPredicates(σ1, this, Sub, (λ σ2 •
  consume(σ2, full, ipre_{Sup_m}, (λ σ3, _ •
    consume(σ3, full, DeptFreeExpr, (λ _ , _, true))
  ) ∧
  produce(σ3[σ2.h / g], fresh, full, ipost_{Sup_m}, (λ σ4 •
    foldPredicates(σ4, this, Sub, (λ σ5 •
      consume(σ5, full, ipost_{Sub_m}, (λ σ6, _ •
        consume(σ6, full, DeptFreeExpr, (λ _ , _, true)))))))))))

where m is a method, Sub its inheriting class, x_1, ..., x_n its parameters, y_1, ..., y_m its return values, (ipre_{Sub_m}, ipost_{Sub_m}) the interface specification of m in Sub and (ipre_{Sup_m}, ipost_{Sup_m}) the interface specification of m in Sup, replacing the implementation specification.

assertInterfaceConformanceOfInheritedFunction(Sub_p, Sup_p) =
let
  γ = {(this, fresh),
       (x_1, fresh), ..., (x_n, fresh),
       (y, fresh)
  }
in
produce(γ, ∅, ∅, {this ≠ null}, ipre_{Sub_p}, (λ σ1 •
unfoldPredicates(σ1, this, Sub, (λ σ2 •
  consume(σ2, full, ipre_{Sup_p}, (λ _ , _ • true))
  ) ∧
  produce(σ2, fresh, full, ipost_{Sup_p}, (λ σ3 •
    foldPredicates(σ3, this, Sub, (λ σ4 •
      consume(σ4, full, ipost_{Sub_p}, (λ _ , _ • true)))))))

where p is a function, Sub its inheriting class, x_1, ..., x_n its parameters, y its return value, (ipre_{Sub_p}, ipost_{Sub_p}) the interface specification of p in Sub and (ipre_{Sup_p}, ipost_{Sup_p}) the interface specification of p in Sup, replacing the implementation specification.
5.5. AUTOMATING THE PROOF OBLIGATIONS

Figure 5.2: Class D inherits a member \( m \) from the class X. The specification refinements indicated ensure that inheriting the member is sound. This is how the interface specification conformance proof obligation has been implemented for inherited members, thereby deviating from the proof rule 4.10. The solid arrows indicate a specification refinement. The thin arrows indicate generation of a specification.

in order to establish a relationship with \( \text{this.valid<Cell>}. \) For the abstract functions this happens automatically.

```java
0 Implementation specification of class Recell {
1     predicate valid // Which is \{ this.valid<Cell> & & acc(bak) \}
2     ...
3
4     // Inherited from Cell which explains the indices
5     method addAlternative(v: int)
6     requires this.valid<Cell>
7     ensures this.valid<Cell>
8     ensures this.get<Cell>() == old(this.get<Cell>()) + v
9 }
10
11 Interface specification of class Recell {
12     predicate valid // Which is \{ this.valid<Cell> & & acc(bak) \}
13     ...
14     // Intended specification uses Recell indices
15     method addAlternative(v: int)
16     requires this.valid<Recell>
17     ensures this.valid<Recell>
18     ensures this.get<Recell>() == old(this.get<Recell>()) + v
19 }
```

Listing 5.1: Implementation and interface specification of class Recell. The full example can be found in appendix A.1

As we did not implement a heuristics automatically exchanging abstract and concrete view of a predicate we have to encode this in the automation by the unfoldPredicates and foldPredicates rules.

The unfoldPredicates takes as input the symbolic heap \( \sigma \), the term of the \( \text{this expression} \), and the inheriting class C. The symbolic heap passed to the continuation
The symbolic execution rules are given in figure 5.3. The foldPredicates operate the other way round. It takes as input the symbolic heap \( \sigma \), the term of the this expression, and the inheriting class \( C \). The symbolic heap passed to the continuation is equivalent to \( \sigma \) except for abstract predicates associated with this and indexed with \( C \) which were unfolded.

The foldPredicates will fold predicates in the symbolic heap \( \sigma.1.h \) as well as in \( \sigma.g \).

The symbolic execution rules are given in figure 5.3.

**Figure 5.3:** Symbolic execution rules for folding and unfolding predicates which are associated with this. The argument subc is a class indicating the abstract predicate family index to use when folding and unfolding. The rule foldPredicates has to fold the predicates on the heap and the old heap. We use the higher-order function foldl as known from Haskell for instance. The first parameter being the set to iterate over, the second parameter is the initial value to use and the third parameter is the combining function. In the combining function we use pattern match to extract the properties of the predicate chunk provided as second argument. We as well use pattern match to extract the predicate chunks from the heap.

```ocaml
unfoldPredicates(\sigma1, t0, subc, Q) =
let
 h10 = \{t0.<subc>[_] # _\} \subseteq \sigma1.h,
in
 Q(
   foldl(h10, \sigma1, (\lambda \sigma, pc @ t.V<subc>[tv] # ta •
   produce(\sigma[[this, t0] / \gamma] - pc, tv, ta, body_{subc,V}, (\lambda \sigma2 •
   (\sigma2 / \sigma.\gamma))))))
where body_{subc,V} is the body of the predicate V in class subc

foldPredicates(\sigma1, t0, subc, Q) =
let
 h10 = \{t0.<>[_] # _\} \subseteq \sigma1.h,
 g10 = \{t0.<>[_] # _\} \subseteq \sigma1.g,
 in
 let
  \sigma g = foldl(g10, \sigma1, (\lambda \sigma, pc @ t.V<> [tv] # ta •
   consume(\sigma / (this, t), ta, body_{subc,V}, (\lambda \sigma2, tv •
   let (hr, \tau r) = merge(\sigma2.h, \{t.V<subc>[tv] # ta\}) in
   (\sigma2 / hr / \sigma.\gamma + \tau r)))))
 in
 Q(
   foldl(h10, \sigma h[\sigma g / g], (\lambda \sigma, pc @ t.V<> [tv] # ta •
   consume(\sigma / (this, t), ta, body_{subc,V}, (\lambda \sigma2, tv •
   let (hr, \tau r) = merge(\sigma2.h, \{t.V<subc>[tv] # ta\}) in
   (\sigma2 / hr / \sigma.\gamma + \tau r))))))
where body_{subc,V} is the body of the predicate V in class subc
```
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

5.5.3 Behavioural Subtyping

The behavioural subtyping proof obligation 4.5 ensures that it is sound to use supertype abstraction when verifying dynamically dispatched calls and function applications. The proof obligation has to be verified by our program verifier and the symbolic execution rules are given in figure 5.4.

One interesting aspect of the rule is that it has to deal with renaming of parameters and return values being the reasons for the numerous fresh terms appearing in the ‘let’ clause.

The symbolic execution succeeds if it is possible to derive the subtypes precondition from the supertypes precondition and this does not leave debts. In a second step we produce the subtypes postcondition which, together with the consumption of its precondition before, abstracts over the execution the subtype’s body. Lastly, we show that the supertype’s postcondition holds without leaving debts.

5.5.4 Incompleteness Arising from Abstract Predicates

The implemented automation of the interface specification conformance proof obligation (see figure 5.1) admits an incompleteness as illustrated with the following listing. The incompleteness occurs only if an abstract predicate family gets indexed explicitly in the specification of a member that in addition gets inherited by another class.

```java
class C {
    var val : int
    predicate valid { acc(val) }
}
class D extends C {
    override predicate valid { valid<C> }
}
class E extends D {
    override predicate valid { valid<D> }
    method unfoldThePredicate ()
        requires valid<E>
        ensures valid<C>
        { unfold valid<E>
            unfold valid<D>
        }
}
class F extends E {
```
assertBehaviouralSubtypingOfMethods(d, c) =
  let
  fresh_1 = fresh, fresh_2 = fresh, ..., fresh_n = fresh
  fresh_{n+1} = fresh, ..., fresh_{n+m} = fresh
  in
  let
    y = {(this, fresh),
          (x^d_1, fresh_1), ..., (x^n_d, fresh_n),
          (x^c_1, fresh_1), ..., (x^n_c, fresh_n),
          (y^d_1, fresh_{n+1}), ..., (y^n_d, fresh_{n+m})
    }
  in
  produce(y, Ø, Ø, {this ≠ null}, tpre_d, (λ σ1 •
    consume(σ1, full, tpre_d, (λ σ2, - •
      consume(σ2, full, DeptFreeExpr, (λ _ • true))
        ∧
      produce(σ2[σ1.h / g], fresh, full, tpost_d, (λ σ3 •
        consume(σ3, full, tpost_c, (λ σ4, - •
          consume(σ4, full, DeptFreeExpr, (λ _ • true))))))))
  )
where d and c are methods, x^d_1, ..., x^n_d respectively x^c_1, ..., x^n_c their parameters, y^d_1, ..., y^n_d respectively y^c_1, ..., y^n_c their return values and (tpre_d, tpost_d) and (tpre_c, tpost_c) be their type specifications.

assertBehaviouralSubtypingOfFunctions(d, c) =
  let
    fresh_1 = fresh, fresh_2 = fresh, ..., fresh_n = fresh
    fresh_y = fresh
  in
  let
    y = {(this, fresh),
          (x^d_1, fresh_1), ..., (x^n_d, fresh_n),
          (x^c_1, fresh_1), ..., (x^n_c, fresh_n),
          (y^d, fresh_y), (y^c, fresh_y)
    }
  in
  produce(y, Ø, Ø, {this ≠ null}, tpre_d, (λ σ1 •
    consume(σ1, full, tpre_d, (λ _ • true))
        ∧
    produce(σ1, fresh, full, tpost_d, (λ σ2 •
      consume(σ2, full, tpost_c, (λ _ • true)))))))
where d and c are function, x^d_1, ..., x^n_d respectively x^c_1, ..., x^n_c their parameters, y^d respectively y^c their return value and (tpre_d, tpost_d) and (tpre_c, tpost_c) be the their specifications.

Figure 5.4: Verification rules for asserting that member d is a behavioural subtype of member c
5.6 Monitor Invariants

We have introduced monitor invariants of noSubclassing-Chalice in section 2.2.2 and outlined the implemented solution in section 3.5.12.

The goal of this chapter is to discuss other solution approaches with their advantages and disadvantages and to detail why the implemented solution is sound.

In the following, we refer to the Chalice statements share, acquire, release and unshare as the monitor statements.

5.6.1 Monitor Invariants and Subclassing

When outlining the implemented solution in section 3.5.12, we remarked that one can consider the monitor invariant as part of the precondition of the share- and release-statements. Furthermore, one can consider the monitor invariants as part of the postcondition of the acquire-statement.

A naive extension of noSubclassing-Chalice would be to use the monitor invariant of the static type for the monitor statements. In this case we observe that the monitor invariants appear in co- and contravariant positions and conclude that the monitor invariant has to be invariant with respect to subclassing in general.

To give an example for the unsoundness that we could encounter otherwise, let $D$ be a subclass of class $C$ and let $d$ be an instance of class $D$. If we share $d$ using static type $C$ and therefore using the monitor invariant of $C$, the monitor invariant of $D$ may not be established. If we then can acquiring $d$ using the static type $D$, we assume the monitor invariant of $D$ which was never established and which is therefore unsound.

However, disallowing subclasses to alter the monitor invariant is restricting. For instance, subclasses may have new state and abstract predicate families which shall be considered in their monitor invariant. The implemented solution allows,
thanks to framing and the restriction that objects can only be shared when knowing its dynamic class, at least to strengthen the monitor invariant in subclasses.

In the implemented approach we allowed non-invariant monitor invariants with respect to subclassing. More precisely, subclasses may add new restrictions to the superclass’ monitor invariant as long as the added restrictions are self-framing. However, in order for that to be sound we have to pay the price that we can only share an object when we establish the monitor invariant of the object’s dynamic type.

5.6.2 Solution Approaches

We present in the following three solution approaches that were not chosen as the one to implement.

- Monitor of dynamic type
- Dedicated predicate family as monitor invariant
- Non-overridable monitor invariants

Monitor Invariant of Dynamic Type

Classes can define/redefine monitor invariants without any restriction. The semantics of the monitor statements is adapted such that they can only be executed when the dynamic type of the affected instance is known (final classes may be of help). The monitor invariant used is the one from the dynamic type.

The limitation of having to know the dynamic type is clearly a restriction. As a consequence of employed factory patterns, the dynamic type may not always be known. Even worse, the dynamic type can be defined in another module and therefore can not be known in general. Programmers may be tempted to check the dynamic type of an instance dynamically which is considered bad design. The implemented solution can be seen as an improved version of the described solution approach as it allows at least acquiring and releasing without knowing the dynamic type.

Dedicated Predicate Family as Monitor Invariant

The idea of this solution approach is to have a predefined monitor invariant monInv for all classes, whereby the predicate family monInv allows subclasses to exchange the body of the predicate.

In order for a client to establish the monitor invariant of an object some additional methods are required. The preShare and postUnshare methods serve as a mean to bring an instance into the state in which it can be shared respectively back to a thread-local state again. The postAcquire and preRelease analogously perform actions to acquire and release an instance. The following listing outlines this idea.
class CellWithMonitorInvariant {
    var val : int

    // Consider the monitor invariant to be fixed
    // and not to be given by the programmer
    invariant monInv

    predicate valid { acc(val) }
    predicate monInv { valid && get() >= 0 }

    method preShare ()
        requires valid && get() >= 0
        ensures monInv
        { fold monInv }
    
    method postUnshare ()
        requires monInv
        ensures valid && get() >= 0
        { unfold monInv }
    
    method postAcquire ()
        requires monInv
        ensures valid && get() >= 0
        { unfold monInv }
    
    method preRelease ()
        requires valid && get() >= 0
        ensures monInv
        { fold monInv }
    
    function get () : int
        requires valid
        ensures true
        { unfolding valid in val }
    
    method set (v : int)
        requires valid
        ensures valid && get() == v
        { unfold valid
          val := v
          fold valid
        }
}

class Program {
    method main ()
        requires true
        ensures true
        { 
            var cell : CellWithMonitorInvariant := new CellWithMonitorInvariant()
            fold cell.valid
            call cell.set(2)
            call cell.preShare()
            share cell

            // cell might be accessed by other threads
The advantage of this solution approach is that subclasses can exchange the body of the \texttt{monInv} predicate. However, as the real monitor invariant manifests itself in the pre- and postconditions of the methods introduced (\texttt{preShare}, \texttt{postAcquire}, \texttt{preRelease}, \texttt{postUnshare}), behavioural subtyping will limit subclasses in their redefinition of the \texttt{monInv} predicate indirectly. Due to this limitation, the repetition of the intended monitor invariant in many pre- and postconditions and the additional methods required, we reject this solution approach.

As a side-remark we would like to emphasize that the presented idea of a dedicated monitor invariant predicate is a good way to reason about monitor invariants in the presence of subclassing.

**Non-overrideable Monitor Invariants**

Sharing an object is only possible if the static type of the shared object or a super type explicitly defines or inherits a monitor invariant. Once a class defines a monitor invariant, subclasses are not allowed to change it syntactically. However, they may indirectly change the monitor invariant by changing bodies of predicates mentioned in the monitor invariant.

The disadvantage is clearly that subclasses are restricted in the way they can change the monitor invariant. For a maximum flexibility a programmer could chose a predicate family only to represent the monitor invariant allowing to exchange the body of the predicate in subclasses. This is exactly the solution approach we discussed in the section before. We therefore do not see any advantage and reject this solution approach.

### 5.6.3 Soundness of the Implemented Solution Approach

We give a proof sketch showing why the implemented semantics for monitor invariants as given in section 3.5.12 is sound. In particular, we outline why it is sound to acquire and release an object without knowing its dynamic type. Recall the notions of effective monitor invariant, defined monitor invariants and in particular that all defined monitor invariants have to be self-framing.
Theorem 5.4  Given the symbolic execution rules from section 4.8.6, acquiring and releasing a shared object of unknown dynamic type $D$ using the monitor invariant of the static type $C$ (which must be a supertype of $D$ or $D$ itself) is sound.

Proof sketch  Let $E_D$ and $E_C$ be the effective monitor invariants of the classes $D$ and $C$ respectively. Let $S_D$ and $S_C$ be sets containing $D$ respectively $C$ together with the superclasses of $D$ and $C$. We then have by the definition of effective monitor invariants that $E_D := \bigwedge_{s \in S_D} I_s$ and $E_C := \bigwedge_{s \in S_C} I_s$ where $I_s$ denotes the defined monitor invariant of a class $s$.

As objects get shared at their dynamic type, it is the effective monitor invariant $E_D$ that is established when acquiring an object and has to be established again when releasing the object.

As $C$ is a supertype of $D$ we can rewrite the definition of $E_D$ as follows.

$$E_D := \left( \bigwedge_{s \in S_C} I_s \right) \land \left( \bigwedge_{s \in S_D \setminus S_C} I_s \right) = E_C \land \left( \bigwedge_{s \in S_D \setminus S_C} I_s \right)$$

When acquiring an object at static type $C$ the monitor invariant $E_C$ gets produced by the symbolic execution of the acquire statement. However, the access permissions contained in the self-framing monitor invariants $\left( \bigcup_{s \in S_D \setminus S_C} I_s \right)$ are not made available and these monitor invariants are preserved until the object is released again.

In the moment the object gets released using the release statement, the symbolic execution rule for this statement ensures that the monitor invariant $E_C$ holds again.

It follows that $E_D$ is established again. $\square$

5.6.4 Sharing an Object without Knowing its Dynamic Type

The implemented solution allows sharing of an object only if its dynamic type is known. In this section we present a design pattern allowing to share an object without knowing its dynamic type. However, this needs explicit wrapping and unwrapping of an object.

Wrapper Design Pattern

The wrapper design pattern demonstrates a way how an instance can be shared between threads without employing a monitor invariant in the class of the shared instance itself. As in example 5.4, a wrapper class is responsible for guaranteeing that the wrapped instance has a certain state when being available in the monitor and that the permissions are transferred between the threads. We present the design pattern in order to support our implementation decision. The design pattern allows to circumvent the requirement of knowing the dynamic class of an object to be shared.
By employing the wrapper pattern, all threads agree on a specific static type at which an instance is shared, acquired and released. The agreed type corresponds to the type used to declare the cell field in the wrapper class. Clearly, there may be arbitrary wrapper classes for sharing at different types available.

```java
final class CellWrapper {
    invariant acc(cell) && cell != null &&
    cell.valid && cell.get() >= 0

    // Note: Instance could be a subclass of Cell
    var cell: Cell
}

class Program {
    var sharedWrapper: CellWrapper

    method main() {
        requires acc(sharedWrapper)
        sharedWrapper = null
        var cell = new Cell(3)
        call shareCell(cell)
    }

    method shareCell(cell: Cell) {
        requires cell.valid && cell.get() >= 0
        requires acc(sharedWrapper) && sharedWrapper == null
        ensures acc(sharedWrapper) && sharedWrapper != null
        ensures acc(sharedWrapper.mu) && waitlevel << sharedWrapper.mu
    }

    share sharedWrapper
    // Note: No need to know the class from which 'cell'
    // was instantiated
    sharedWrapper.cell := cell
    // was instantiated as the CellWrapper-class
    // is declared to be final.
}
```

Listing 5.4: Example demonstrating the design pattern. The Cell class is as given in listing A.1 of the appendix.
Since this master’s thesis is part of a project with the long-term goal of developing a verifier for Scala [18], we investigated how to apply or extend the presented verification approach in order to verify Scala traits.

In the following we will give a brief introduction to Scala traits and related concepts, outline how to use the developed formalism to support traits and describe the remaining challenges.

The Chalice version with support for traits will be referred to as traits-Chalice. However, traits-Chalice has not been implemented as part of this work.

### 6.1 Scala Traits

Scala traits can be understood as interfaces with implementation and state. Their purpose is to allow fine-grained code reuse.

Scala traits can provide fields, abstract method declarations and method definitions. A Scala trait extends a class, may mixin other Scala traits but may not be instantiated itself. Scala classes may extend a single class, but may mixin multiple traits. We denote the combination of a class with traits as a *mixin composition*.

### 6.2 Related Concepts

The semantics of Scala traits is similar to *traits* as defined by Schärli et al [26] and *flavors* (also known as mixins) introduced by Moon [17]. However, Scala traits, do despite their name, differ in at least two important aspects from traits as defined by Schärli et al. First of all, Scala traits can have state as opposed to Schärli traits. Secondly, conflicts arising from a mixin composition of Scala traits providing overriding methods with the same signature are resolved implicitly by giving a linearization order on the Scala traits. Schärli traits force the programmer to resolve the conflict explicitly. Thus, while the composition order is irrelevant for Schärli traits it is relevant for Scala traits. After all, Scala traits are not Schärli traits but rather flavors as introduced by Moon. Similar to Scala traits, flavors have state
and an ordering similar to the linearization order of Scala traits. However, flavors give the programmer a more detailed control over how methods are inherited.

In the following chapters we refer with traits to Scala traits.

### 6.3 Motivating Example

In the following we distinguish two use cases of traits and give, for each use case, an illustrating example.

The potential use cases of traits has been studied by [25]. As we will restrict ourselves to traits without abstract members, several use cases get ruled out.

One use case we consider are traits that enrich the interface of their superclass. Such traits typically do not provide own state as they merely provide new functionality to manipulate the state of the superclass. This can be seen as an instance of a trait providing a rich interface for a class.

The other use case we look at are traits introducing new side-effects to overridden methods of the superclass.

For the discussion of the motivating examples we will make use of an adapted example of the Cell class we have introduced in listing 3.1. We call the adapted Cell class BasicCell. The BasicCell consists of the val field, the valid predicate, the set method and the get function but not the other members. The example is given in the appendix B.1.

#### 6.3.1 Traits Enriching the Interface

We give two definitions of traits and one definition of a class. Concerning the following two definitions of the traits Swapping and Adding, observe that the provided methods are external methods (i.e have the dynamic modifier). This seems to be typical for traits enriching the interface of a class as the traits only use the interface of the superclass and do not access state directly. As a side-remark: Note that we do not consider a trait that accesses state of the superclass directly to enrich the interface but to add new features.

```scala
trait Swapping extends BasicCell {
  method dynamic swap(other: BasicCell)
  requires valid
  requires other != null && other.valid
  ensures valid
  ensures other.valid
  ensures this.get() == old(other.get())
  ensures other.get() == old(this.get())
  {
    var tmp := other.get()
    call other.set(get())
    call set(tmp)
  }
}
trait Adding extends BasicCell {
```


6.3. MOTIVATING EXAMPLE

The following listing defines a class AddingAndSwappingCell mixing-in the traits Adding and Swapping. That is, the resulting interface contains the function get and the methods set, add and swap. The verification approach presented in the remainder of this chapter is expected to be able to verify the given traits and the class AddingAndSwappingCell.

```java
class AddingAndSwappingCell extends BasicCell
    with Adding with Swapping
{
    // All members inherited
}
```

Listing 6.2: Mixin composition involving the Swapping and Adding traits. The contract AddingAndSwappingCellContract is given in appendix B.10.

### 6.3.2 Traits Introducing Side-Effects

We give two definitions of traits and two definitions of classes.

The BackupKeeper trait implements the same semantics as the Recell class we have presented earlier in listing 3.1. The InvocationCounter trait counts the number of times the value of the BasicCell has been set.

Observe that both traits override the set method of the BasicCell class. In addition, observe that both traits add new state.

In the predicate definition, we use super.valid to indicate that we mean the abstract predicate of the super class.

```java
trait BackupKeeper extends BasicCell {
    var bak: int

    override predicate valid { super.valid && acc(bak) }

    function getBak(): int
        requires valid
    { unfolding valid in bak }

    override method set(v: int)
        requires valid
    ensures valid && get() == v
    ensures getBak() == old(get())
    { }
```

Listing 6.1: Trait examples enriching the interface of class BasicCell.
The following listing defines two classes mixing-in the traits `BackupKeeper` and `InvocationCounter` in a different order. However, as specified by the contract
6.4. ADDING TRAITS TO CHALICE

In this section we outline how to adapt subclassing-Chalice in order to support a subset of Scala traits. At the same time we explain the semantics of traits although we assume the reader to be already familiar with Scala traits to some extent. We will also introduce the new language concepts needed for verification.

6.4.1 Contract Definition

A contract serves as a mean for the programmer to provide an interface specification to a mixin composition. In particular useful are contracts when two traits in a mixin composition override the same method of their superclass and, for instance, strengthen the postcondition in different ways. Observe that this is the case for the mixin compositions BackupCountigCell and CountingBackupCell in listing 6.5 with respect to the set method of the mixed-in traits. The question is what the
resulting specification for the set method in the mixin composition shall be. A contract allows the programmer to resolve the conflict and to state the intended (e.g combined) postcondition, as done for the mentioned example in listing 6.4.

While we assume in the following that the programmer provides the contracts separately, one could allow the programmer to provide them just at the mixin site. A contract has a name and may refine other contracts. A contract not mentioning another contract to refine implicitly refines the special AnyRefContract corresponding to the special AnyRef class. In the current implementation, the AnyRef contract does not contain any members. A contract gets declared as

contract M refines N₁, N₂, ..., Nₙ

where M is the name of the declared contract and N₁, N₂, ..., Nₙ with n ≥ 0 are other contracts. A contract may get refined by several other contracts. However, a contract must not refine itself or another contract such that it would refine itself indirectly.

A contract contains field, predicate, method, function declarations but no predicate, method or function definitions. Method and function declarations are equipped with a specification as in subclassing-Chalice. Furthermore, a contract does not contain constructor declarations. The following listing provides an example of a contract.

```
contract BasicCellContract {
  var val: int
  predicate valid
  function get(): int
    requires valid
    ensures true
  method set(v: int)
    requires valid
    ensures valid && get() == v
}
```

Listing 6.6: Contract for the BasicCell class from listing 3.1.

If a contract D refines another contract C this expresses that D describes a behavioural subtype of C. That is for each member in C, there is a member in D with the same signature, contravariant parameter- and covariant return-types.

The idea of contracts is adopted from Schwerhoff [25]. The specification given in a contract will be referred to as code specification in addition to the code specification of classes we already introduced earlier.

### 6.4.2 Trait Definition

A trait has a name, extends a class and may provide a list of other traits to be mixed in. Thus, a trait gets declared as

```
trait T extends S with T₁ with ... with Tₙ module U
```
where $S$ denotes a class, $T_1, \ldots, T_n$ with $n \geq 0$ denote other traits and $U$ denotes a module. The extends clause may be omitted and defaults to extending the AnyRef class.

A trait may not be instantiated itself. The definition of a trait may contain field, method, function and predicate definitions as known from classes in subclassing-Chalice. The member bodies may contain super calls and may access or override members of the superclass $T$ or the mixed-in traits $T_1, \ldots, T_n$. In addition, despite the fact that traits can not be instantiated, traits may define a constructor. However, constructors in traits have additional restrictions compared to constructors as in subclassing-Chalice.

**Constructors**

Traits can specify a primary constructor but no auxiliary constructors as classes are allowed to. Furthermore, the primary constructor of a trait is neither allowed to take parameters nor to pass values to a super type’s constructor. Otherwise, the semantics and syntax of constructors in trait-Chalice are as in subclassing-Chalice.

As in subclassing-Chalice, constructors serve as a source of permissions for the state added by the trait. If the programmer does not provide a constructor explicitly, a default constructor gets generated.

An open challenge is with respect to the specification of constructors. The challenge gets discussed in section 6.10.2.

**6.4.3 Class Definition**

A class declaration in traits-Chalice is given as

```
class D extends C with T_1 with \ldots with T_n respects M
```

where $S$ denotes a class, $T_1, \ldots, T_n$ with $n \geq 0$ denote traits and $M$ denotes a contract. The extends clause may be omitted and defaults to extending the AnyRef class.

The definition of a class is as in subclassing-Chalice, except with the addition that member bodies may access members of the traits $T_1, \ldots, T_n$.

For simplicity, we force every class declaration in traits-Chalice to provide a respects clause\(^1\). Furthermore, whenever a class $D$ extends a class $C$, then the contract $M$ of $D$ has to refine the contract $N$ of $C$. Treatment in the following gets simpler as we can use the contracts to enforce behavioural subtyping.

\(^1\)An implementation would try to generate the contract automatically whenever possible from the individual specifications. That is, the contract of a class would refine the contract of the specified superclass, if any, and the specification would be generated from the class itself and its mixed-in traits.
6.4.4 Anonymous Mixin Compositions

A program may contain anonymous mixin compositions. An example is given by the following declaration,

\[
\text{method meth}(o: S \text{ with } T_1, \ldots, T_n \text{ respects } M)
\]

where S denotes a class, \( T_1, \ldots, T_n \) with \( n \geq 0 \) denote traits and \( M \) denotes a contract.

Anonymous mixin compositions allow to specify a mixin composition without involving an additional class glueing the composed class and traits together.

For simplicity, we force every anonymous mixin composition declaration in traits-Chalice to provide a \texttt{respects} clause.

6.4.5 Templates

A so-called template 'S with \( T_1 \) with ... with \( T_n \)', as it appears in the declaration of a trait, class or anonymous mixin composition, consists of the superclass \( S \) and trait references \( T_1, \ldots, T_n \).

We state the criteria for well-formedness according to the Scala specification. Templates that are not well-formed shall be rejected by the type checker. In the following, we assume that templates that we state in examples are always well-formed.

**Definition 6.1** A template \( S \text{ with } T_1 \text{ with } \ldots \text{ with } T_n \) is well-formed if the superclass \( S \) is a subclass of all the superclass of \( T_1, \ldots, T_n \).

6.4.6 Linearization Order

As in subclassing-Chalice, classes may override methods and functions of their superclass. Thereby, a conflict may arise when traits of the same mixin composition override the same method or function in the superclass (see figure 6.1). We implement the Scala solution which is to provide a linearization order on the traits and classes involved in a mixin composition. The linearization order of the mixin composition defines the order in which overriding takes place and is as well used to bind super member invocations.

An example of a linearization order was given in listing 6.5. We observe the linearization of the class \texttt{CountingBackupCell} which is given by: \texttt{CountingBackupCell, BackupKeeper, InvocationCounter, BasicCell, AnyRef}. Given this linearization, the \texttt{set} method of \texttt{BackupKeeper} overrides the \texttt{set} method of \texttt{InvocationCounter}, which in turn overrides the \texttt{set} method of \texttt{BasicCell}. Thereby, the conflict resulting from both traits overriding the \texttt{set} method of \texttt{BasicCell} gets resolved.

We give the formal definition of the linearization order as given in the Scala reference [18].
6.4. ADDING TRAITS TO CHALICE

Definition 6.2 Let $C$ be a class or trait with template $S$ with $T_1, \ldots, T_n$. The linearization of $C$, $\mathcal{L}(C)$, is defined as follows:

$$\mathcal{L}(C) = C, \mathcal{L}(T_n) \boxplus \ldots \boxplus \mathcal{L}(T_1) \boxplus \mathcal{L}(S)$$

The $\boxplus$ operator denotes concatenation whereby elements of the right operand replace identical elements of the left operand. In the following, ‘a’ is the head and ‘A’ the tail of the linearization order $(a, A)$.

$$(a, A) \boxplus B = \begin{cases} a, (A \boxplus B) & \text{if } a \notin B \\ A \boxplus B & \text{if } a \in B \end{cases} \quad (6.1)$$

Properties of the Linearization Order

Figure 6.2 gives an example of classes and traits extending each other and mixing-in each other. Observe that the resulting directed graph describes a partial order, denoted as $PO$, on traits and classes. Assume that all this classes and traits are involved in a mixin composition $M$. The task of the linearization order of the mixin composition $M$ is to define a unique total order on the traits and classes satisfying the partial order $PO$. That is, the linearization order refines the partial order $PO$. The following properties for the linearization order can be derived, which are as well given in the Scala specification.

The linearization order has the following properties:

- The linearization order of a class refines the subclassing relation. That is, if $D$ is a subclass of $C$, then $D$ precedes $C$ in any linearization where both $D$ and $C$ occur.
- Let $\mathcal{L}(D)$ be linearization of a class $D$ with direct superclass $C$. The linearization $\mathcal{L}(C)$ is a suffix of $\mathcal{L}(D)$.
- Let $\mathcal{L}(AMC)$ be linearization of an anonymous mixin composition with superclass $C$. The linearization $\mathcal{L}(C)$ is a suffix of $\mathcal{L}(AMC)$.
6.4.7 Late Bound Super Member Invocations

In subclassing-Chalice, super member invocations (i.e super calls and super function applications) have been bound statically. In the last section we introduced the linearization order for mixin compositions in order to resolve the conflict when two traits override the same method of their superclass. Taking up this example again, if a trait overrides a method of the superclass and uses a super member invocation inside the new body then it follows that we have to use the linearization order as well to bind super member invocations.

As the linearization order depends on a concrete mixin composition, the class or trait to which the super reference inside a trait gets bound is not defined for a trait alone. Only after the trait gets involved in a mixin composition the super reference can be bound for that particular mixin composition. We refer to this issue with late bound super invocations or simply late bound super calls.

An example is given by the traits BackupKeeper and InvocationCounter in listing 6.3. Both traits use a super call for which we do not know to which class or trait super refers to. The super call gets bound on mixin composition time, that is for example when the traits get mixed-into a class as it happens in listing 6.5. Looking at the class CountingBackupCell, the super call inside the set method of trait BackupKeeper gets bound to the implementation provided by the trait InvocationCounter.

6.4.8 Deviation from Scala Traits

We deviate in two aspects from Scala traits as discussed in the following.
6.4. ADDING TRAITS TO CHALICE

Abstract Members

In contrast to Scala traits, we restrict us to traits without abstract members. The reason is merely simplicity of the supported language. As a consequence of not supporting abstract members, we lose the ability to use traits in the sense of Java interfaces for example.

Symmetric Traits in Type System

A further simplification we assume with respect to the type system. We describe Scala’s implementation of the subtype relation: A mixin composition \( D \) is a subtype of another mixin composition \( C \), if \( D \) contains at least the classes and traits of \( C \). That is, Scala allows \( D \) to have traits reordered to some extent. We give an example.

```
0  class S {}
1  trait A extends S {}
2  trait B extends S {}
3  class AB extends S with A with B {}
4  class BA extends S with B with A {}
5
6  var ab : AB = new BA()
```

Listing 6.7: An example what Scala’s type system allows

We use a more restrictive type system. That is, in addition to the requirements given by Scala, we require an additional property with respect to the contracts.

**Definition 6.3** A mixin composition \( D \) with associated contract \( M \) is a subtype of another mixin composition \( C \) with associated contract \( N \) if and only if \( D \) is a subtype of \( C \) according to the rules of Scala [18] and in addition the contract \( M \) refines directly or indirectly contract \( N \).

6.4.9 Definitions

We give some more definitions needed for the following discussion.

We refer to a **mixin composition** for classes having no traits mixed in, classes having traits mixed in, and anonymous mixin compositions.

Given a linearization order \( L_1, L_2, \ldots, L_i, L_{i+1}, \ldots, L_n \) where \( L_n = \text{AnyRef} \), the **successor** of a trait or class \( L_i \) in a linearization order is the class or trait \( L_{i+1} \).

The **super class or trait** of some trait is its successor in the given linearization order.

The **dynamic mixin composition** of a reference is the mixin composition that was used to instantiate the referred object.

Given a linearization order \( L_1, L_2, \ldots, L_i, L_{i+1}, \ldots, L_n \) where \( L_n = \text{AnyRef} \), the **static class or trait** of a \texttt{this}-expression appearing inside a class or trait \( L_i \) is \( L_i \). Furthermore, the **static class or trait** of a \texttt{super}-expression appearing inside a class or trait \( L_i \) is \( L_{i+1} \).
6.4.10 Abstract Families and Traits

We have to adapt the abstract families in order to work with traits. The only relevant change is that abstract families in traits-Chalice can be indexed by classes and traits. We do not restate the definitions of abstract functions and predicates and their families. We only restate the definition for the dynamic symbol for abstract predicate families and abstract function families.

Definition 6.4 An abstract predicate family \( \text{rcvr}.V \) indexed by the special symbol \text{dynamic}, that is \( \text{rcvr}.V<\text{dynamic}>, \) expresses that the abstract predicate \( \text{rcvr}.V<L_1> \) is meant, where \( L_1 \) is the first element in the linearization order \( \mathcal{L}(D) = L_1, \ldots, \text{AnyRef} \) of the dynamic mixin composition \( D \) of the receiver \( \text{rcvr} \). Furthermore, an abstract predicate indexed with the \text{dynamic} symbol does only have an abstract view.

Definition 6.5 An abstract function family \( \text{rcvr}.p(a_1, a_2, \ldots, a_n) \) indexed by the special symbol \text{dynamic}, that is \( \text{rcvr}.p<\text{dynamic}>(a_1, a_2, \ldots, a_n) \), expresses that the abstract function \( \text{rcvr}.p<L_1>(a_1, a_2, \ldots, a_n) \) is meant, where \( L_1 \) is the first element in the linearization order \( \mathcal{L}(D) = L_1, \ldots, \text{AnyRef} \) of the dynamic mixin composition \( D \) of the receiver \( \text{rcvr} \). Furthermore, an abstract function indexed with the \text{dynamic} symbol does only have an abstract view.

Similar to subclassing-Chalice, the dynamic symbol represents the class or trait appearing first in the linearization order of the dynamic mixin composition of the receiver. In subclassing-Chalice, this was simply the dynamic class of the receiver as there were no traits.

6.4.11 Inheritance of Members

Given a mixin composition, predicates as well as other members are inherited by aggregation along the linearization order of the mixin composition analogously to subclassing-Chalice (section 3.5.8). That is, given a linearization order \( L_1, \ldots, L_m, \text{AnyRef} \), a trait or class \( L_i \) inherits all members of all classes or traits \( L_j \) with \( j > i \) except for the members overridden by \( L_i \).

6.5 Verification Goal

We aim for a modular verification schema for traits-Chalice. In particular, we want to verify the bodies of members given in a trait once only.

The non-modular approach would be to verify the bodies of trait members for each mixin composition once. It is non-modular as mixin compositions can occur in a module that is different from the modules of the traits involved. That is, accessing the trait bodies for reverification is not always possible.

In contrast to subclassing-Chalice, we will not support the respecification of methods and functions in traits-Chalice. The reason is that the respecification com-
6.6 Deriving the Verification Specifications

With the introduction of traits we have to detail how the verification specifications get generated.

For subclassing-chalice we have generated the verification specifications for classes. In traits-Chalice we generated the verification specifications for mixin compositions. For traits we do as well generated the implementation specification but not the interface specification and type specification, as traits can not be instantiated.

Figure 6.3 outlines the new setting.

Figure 6.3: A mixin composition is always associated with a contract and consists of a class (including its superclasses) and an arbitrary number of traits mixed-in. Recall that classes are considered to be mixin compositions themselves. The contract is used to generate the interface and type specification. The implementation specification of a mixin composition depends on the linearization order. For traits only an implementation specification gets generated as they can not be instantiated.

In the following we will refer with the term members to methods, functions and constructors.

6.6.1 Deriving the Implementation Specification

We have to describe how to derive the implementation specification of an entire mixin composition as well as how to derive it for a specific member defined in a class or trait. We denote the transformation described in the following as implementation specification transformation.

We give the rule for a single member first. The implementation specifications of a member results from the code specification by providing indices for the abstract families. Otherwise, the two specifications are syntactically identical.
Let $C_m$ denote a code specification of a member $m$ introduced or overridden in class or trait $C$ and let $B_m$ be the implementation specification of $m$. The indices for abstract families (predicates and functions) are selected as follows.

1. If $m$ is an external member, then all indices are dynamic.
   That is $B_m := I(C_m, \text{dynamic})$

2. If $m$ is an internal member, then use the syntactical indexing rule.
   That is $B_m := SI(C_m, C)$

Given all implementation specifications of members, we can give the rule for composing the implementation specification of a mixin composition $MC$. Let the linearization order of the mixin composition be $L(MC) = L_1, L_2, \ldots, L_n, \text{AnyRef}$.

1. Let $DM$ be the set of implementation specifications of members introduced or overridden in class or trait $L_1$.

2. Let $IM$ be the set of implementation specifications of members introduced or overridden in $L_2, \ldots, L_n, \text{AnyRef}$ but not overridden in $L_1$. Imprecisely, this are the implementation specifications of the inherited members.

3. The implementation specification $B$ of mixin composition $MC$ is given as $B := DM \cup IM$

The implementation specification of a trait is composed only of the members defined or overridden in the trait. That is, while the implementation specification of a mixin composition contains the specifications of inherited members the implementation specification of a trait does not. The reason is that for traits we do not know statically which members we inherit.

The implementation specification of our running examples can be found in the appendix B.

### 6.6.2 Deriving the Interface Specification

The interface specification gets generated from the contract associated with a mixin composition. As traits do not have an associated contract and can not be instantiated, we do not generate an interface specification for traits. We denote the transformation described in the following as interface specification transformation.

Let $MC$ be a mixin composition with linearization order $L_1, L_2, \ldots, L_n, \text{AnyRef}$ and $N$ its associated contract. Let $N_m$ denote the code specification of a member $m$ in $N$. The interface specification $I$ for $MC$ results from the code specification by providing indices for the abstract families. Otherwise the two specifications are syntactically identical. We therefore only give the rule how to index the abstract families.

1. If $m$ is an external member, then all unspecified indices are dynamic.
   That is $I_m := I(N_m, \text{dynamic})$

2. If $m$ is an internal member, then use the syntactical indexing transformation.
   That is $I_m := SI(N_m, L_1)$
The interface specification of our running examples can be found in the appendix B.

### 6.6.3 Deriving the Type Specification from the Contract

The type specifications get generated from the contract associated with the mixin composition in an analogous way as the interface specifications get generated. We denote the transformation described in the following as *type specification transformation*.

Let $MC$ be a mixin composition and $N$ its associated contract. Let $N_m$ denote the specification of a member $m$ in $MC$. The type specification $T$ for $MC$ results from the code specification by providing indices for the abstract families. Otherwise the two specifications are syntactically identical. We therefore only give the rule how to index the abstract families.

$$T_m := \mathcal{I}(N_m, \text{dynamic})$$

The type specification of our running examples can be found in the appendix B.

### 6.7 Adapting presented proof obligations

This section is dedicated to restate the proof obligations from chapter 4 taking traits into account. We do neither motivate nor explain the proof obligation again and refer to chapter 4 for that. However, we will explain the changes made to them. Figure 6.4 and figure 6.5 outline the new setting.

![Diagram](image)

**Figure 6.4:** A single mixin composition with its contract and the specification refinement enforced in order to ensure that inheriting members and mixing-in traits is sound. The solid arrows indicate a specification refinement. The black-headed arrows indicate generation of a specification.

#### 6.7.1 Type Specification Conformance

We have to adapt the type conformance proof rule and obligation in order to take care of mixin compositions. Mixin compositions take the place of the classes when compared to the proof obligations for subclassing-Chalice.
We need to ensure that the type specification of a mixin composition $MC$ refines the interface specification of mixin composition $MC$, given that the effective instance bound to this is mixin composition $MC$.

We use $\text{dynamicMixinComposition(this)} = MC$ to denote that the instance this refers to is an instance of $MC$.

**Proof rule 6.6** Type conformance of a member $m$ of a mixin composition $MC$.

Let the interface specification of $m$ be $(I, I')$ and the type specification be $(T, T')$. The proof rule is satisfied if the type specification refines the interface specification of the member $m$ given the additional assumption that the dynamic mixin composition of this is $MC$.

Formally,

$$T \vdash \{I\}, \{I'\} \Rightarrow \{T \land \text{dynamicMixinComposition(this)} = MC\}, \{T'\}$$

**Proof obligation 6.7** Let $P$ be the program to verify. All methods and functions of all mixin compositions of $P$ have to satisfy the type conformance proof rule 6.6.

### 6.7.2 Behavioural Subtyping

As the type system uses contracts to enforce behavioural subtyping, we need to check that the refinement relation induced by the contracts guarantee behavioural subtyping. Recall that the type specifications get generated from the contracts.

**Proof obligation 6.8** Let $P$ be the program to verify. For each contract $M$ in $P$ declaring to refine another contract $N$ we have to check behavioural subtyping. That is, for every member $n$ for which $N$ contains a code specification, there must be a corresponding member $m$ in $M$ such that $m$ is a behavioural subtype of $n$ according to the behavioural subtyping proof rule 4.4.
6.8. DEALING WITH LATE BOUND SUPER INVOCATIONS

6.7.3 Interface Specification Conformance

In contrast to subclassing-Chalice, the interface specification of a mixin composition (which includes classes) gets generated from its associated contract. We have to check that the interface specification refines the implementation specification.

Proof obligation 6.9 Let $P$ be the program to verify. For each mixin composition (i.e. anonymous mixin compositions or classes) appearing in $P$, the interface specification must refine the implementation specification. That is, for any member $m$ of a mixin composition, the interface conformance proof rule 4.10 has to hold.

6.8 Dealing with Late Bound Super Invocations

When verifying a trait modularly we do not know at which position it appears in the linearization order of a mixin composition. A direct consequence is that we can not determine where super member invocations get bound to. We have discussed late bound super invocations already in section 6.4.7. For the verification, late bound super invocations can be treated similarly to dynamically bound member invocations where we also do not know to which members the invocation gets bound.

For the verification of late bound super invocations in traits we use a kind of supertype abstraction which we call in this context static superclass abstraction. That is, we introduce a new kind of specification, denoted static specification, abstracting over the exact super class or trait and enforce certain specification refinements similar to behavioural subtyping in order to ensure that our abstraction is sound.

We first present the idea of static superclass abstraction and then define the static specification.

6.8.1 Static Superclass Abstraction

The idea of static superclass abstraction is to use the static specification of the traits’ superclass in order to verify super member invocations. The superclass thereby serves as an abstraction over the potential super classes and traits that could get bound to $\text{super}$ within a trait of a concrete mixin composition.

To be precise, we give a formal example. Let $T$ be a trait to verify and let $Q$ be the unknown class or trait to which a particular invocation on $\text{super}$ would get bound inside $T$, within a concrete mixin composition. Furthermore, let $\text{SC}$ be the specified superclass of $T$. In order to verify super member invocations inside $T$, we used the static specification of $\text{SC}$. Thereby, the superclass $\text{SC}$ serves as an abstraction over all possible $Q$’s.

In order for static superclass abstraction to be sound, we have to ensure that the static specification of the superclass $\text{SC}$ refines the static specification of all possible $Q$’s.
6.8.2 Static Specifications

For each class a static specification gets generated from the associated contract. The static specification of a class is up to indices to abstract families identical to the interface specification of the same contract. The purpose of the static specification is to abstract over the actual class the specification belongs to. This allows to use the static specification of the superclass in place of the unknown specification super gets bound to when verifying super member invocations inside a trait.

The static specification generated for a trait is up to indices to abstract families identical to the implementation specification of the trait.

Figure 6.6 outlines how the specifications are generated.

Figure 6.6: Static specifications of classes and traits. The thin and black-headed arrows indicate generation of a specification.

The Static Symbol

In order to abstract over the concrete class or trait a static specification belongs to, we need to introduce a special symbol \texttt{static} with a similar idea as the symbol \texttt{dynamic} we already introduced for subclassing.

\textbf{Definition 6.10} An abstract predicate family \texttt{rcvr.name} indexed by the special symbol \texttt{static} expresses that the abstract predicate \texttt{rcvr.name\texttt{S}} is meant, where \texttt{S} is the static class or trait of the associated instance \texttt{rcvr} in a concrete mixin composition. The \texttt{static} index may only be used when the \texttt{rcvr} is syntactically identical to \texttt{this} or \texttt{super}.

Furthermore, an abstract predicate indexed with the \texttt{static} symbol does only have an abstract view.

\textbf{Definition 6.11} An abstract function family \texttt{rcvr.name(pars)} indexed by the special symbol \texttt{static} expresses that the abstract function \texttt{rcvr.name\texttt{S}(pars)} is meant where \texttt{S} is the static class or trait of the associated instance \texttt{rcvr} in a concrete mixin composition. The \texttt{static} index may only be used when the \texttt{rcvr} is syntactically identical to \texttt{this} or \texttt{super}.

Furthermore, an abstract function indexed with the \texttt{static} symbol does only have an abstract view.
To give an example for the static symbol: Within a trait $T$ and given a concrete mixin composition the trait is involved, let super refer to the trait SuperTrait and let this trait inherit the predicate valid from its superclass. The expression super.valid$<\text{static}>$ (or for short super.valid) within trait $T$, given the concrete linearization, is equivalent to this.valid$<\text{SuperTrait}>$ as well as super.valid$<\text{SuperTrait}>$ (i.e. the static type of the receiver only matters for the static symbol). Furthermore, inside the static specification of SuperTrait the expression this.valid$<\text{static}>$ is equivalent to this.valid$<\text{SuperTrait}>$.

**Example for a Static Specification**

The following example gives the generated static specification of the BasicCell class and the two traits BackupKeeper and InvocationCounter. Observe that the abstract families are not indexed with concrete classes but with symbols only.

```
0  Static specification of class BasicCell {
1    var val: int
2
3    predicate valid
4
5    function get(): int
6      requires this.valid$<\text{static}>$
7      ensures true
8
9    method set(v: int)
10      requires this.valid$<\text{static}>$
11      ensures this.valid$<\text{static}>$
12      ensures this.get$<\text{static}>() == v
13  }

Listing 6.8: Static specification of class BasicCell from listing B.1.
```

```
0  Static specification of trait BackupKeeper {
1    var bak: int
2
3    predicate valid
4
5    function getBak(): int
6      requires this.valid$<\text{static}>$
7
8    method set(v: int)
9      requires this.valid$<\text{static}>$
10      ensures this.valid$<\text{static}>$
11      ensures this.getBak$<\text{static}>() == \text{old}(this.get$<\text{static}>())
12      ensures this.getBak$<\text{static}>() == old(this.get$<\text{static}>())
13  }

Listing 6.9: Static specifications of the trait BackupKeeper.
```

```
0  Static specification of trait InvocationCounter {
1    var counter: int
2
3    predicate valid
4
5    function getCounter(): int
6      requires this.valid$<\text{static}>$
7
8    method set(v: int)
9      requires this.valid$<\text{static}>$
10      ensures this.valid$<\text{static}>$
11      ensures this.get$<\text{static}>() == v
12      ensures this.get$<\text{static}>() == old(this.get$<\text{static}>())
13  }
```

Listing 6.10: Static specifications of the trait InvocationCounter.
CHAPTER 6. TRAITS

Listing 6.10: Static specifications of the trait InvocationCounter.

Verification Example

We give an example how verification proceeds for the set method of the InvocationCounter trait given in appendix B.16.

In the following we restate the body of InvocationCounter.set and provide comments and assert statements indicating how the verification proceeds.

Listing 6.11: Example verification using static superclass abstraction. The provided set method is from trait InvocationCounter as given in appendix B.16. We only outline the verification aspects related to the super function application.

6.8.3 Derivation of the Static Specification

We state how to derive the static specification of classes and the static specification of traits.
Static Specification of Classes

The static specification of a class gets generated from the contract associated with the class.

Let $N$ be the contract associated with a class $C$. Let $N_m$ denote the specification of a member $m$ introduced, overridden or inherited in $C$. The static specification $S_m$ for $m$ results from by providing indices for the abstract families. Otherwise the two specifications are syntactically identical. We therefore only give the rule how to index the abstract families.

1. If $m$ is an external member, then all unspecified indices are dynamic.
   That is $S_m := \mathcal{I}(N_m, \text{dynamic})$

2. If $m$ is an internal member, then use the syntactical indexing transformation.
   That is $S_m := S\mathcal{I}(N_m, \text{static})$

The reason for indexing external members with dynamic is that the dynamic symbol abstracts over the dynamic mixin composition of an instance and not over a particular class or trait the specification belongs to. Therefore, there is no need to index with static.

Static Specification of Traits

For traits we do not have a contract in the sense of section 6.4.1 available and the interface specification is not defined for a trait as they can not be instantiated. We derive the static specification therefore directly from the code specification of the trait.

Let $T$ be the trait and $B$ be its code specification. Let $B_m$ denote the code specification of a member $m$ introduced or overridden in $T$. The static specification $S_m$ for $m$ results from by providing indices for the abstract families. Otherwise, the two specifications are syntactically identical. We therefore only give the rule how to index the abstract families.

1. If $m$ is an external member, then all unspecified indices are dynamic.
   That is $S_m := \mathcal{I}(B_m, \text{dynamic})$

2. If $m$ is an internal member, then use the syntactical indexing transformation.
   That is $S_m := S\mathcal{I}(B_m, \text{static})$

6.8.4 Proof Obligations

The two proof obligations needed are analogous to the proof obligations type conformance and behavioural subtyping as presented section 4.6. In the following we present the new proof obligations.

Static Specification Conformance

We want to relate the static specification we use for abstraction to the implementation specification.
For traits we do this directly by requiring that the static specification of a trait refines the implementation specification of the trait. For classes we do it indirectly by requiring the static specification of class to refine its interface specification. Concerning the proof rule for classes, we could use the implementation specification in place of the interface specification. However, as the static specification and the interface specification of a class are generated in a similar fashion, automation of the specification refinement is simpler when using the interface specification in the proof rule.

In figure 6.7 we indicate the mentioned specification refinements. We state the proof rules and the proof obligation in the following.

![Diagram of specification refinements]

**Figure 6.7**: The specification refinements for a class and a trait for the shown specifications. The solid arrows indicate a specification refinement. The thin and black-headed arrows indicate generation of a specification.

**Proof rule 6.12** Static specification conformance of a member \( m \) of trait \( T \).
Let the implementation specification of \( m \) be \((B, B')\) and the static specification be \((S, S')\). The proof rule is satisfied if the following specification refinement holds.

\[
T \vdash \{B\} \rightarrow \{B'\} \Rightarrow \{S \land \text{this:}C\} \rightarrow \{S'\}
\]

whereby \( \text{this:}C \) denotes that the static trait of \( \text{this} \) is \( C \).

**Proof rule 6.13** Static specification conformance of a member \( m \) of class \( C \).
Let the interface specification of \( m \) be \((I, I')\) and the static specification be \((S, S')\). The proof rule is satisfied if the following specification refinement holds.

\[
T \vdash \{I\} \rightarrow \{I'\} \Rightarrow \{S \land \text{this:}C\} \rightarrow \{S'\}
\]

whereby \( \text{this:}C \) denotes that the static class of \( \text{this} \) is \( C \).

**Proof obligation 6.14** Let \( P \) be the program to verify. All methods and functions of all classes in \( P \) have to satisfy the static specification conformance proof rule 6.12. In addition, all methods and functions of all traits in \( P \) have to satisfy the static specification conformance proof rule 6.13.

**Behaviour Preservation**

For static superclass abstraction to be sound we need to ensure that traits and classes override methods and functions in way that preserves the behaviour of the
overridden member. This is what the following proof rules and proof obligations are concerned with. We need two proof obligations, one for classes and one for traits.

The following proof rule is identical to proof rule 4.4 except that it works on the static specification instead of the type specification. We keep them separate for a simpler presentation and to separate their concerns explicitly.

**Proof rule 6.15** Behaviour preservation proof rule for a member ‘d’ overriding a member ‘c’.

Let \((S_c, S'_c)\) and \((S_d, S'_d)\) be the static specifications of \(c\) and \(d\), respectively. The proof rule is satisfied if the static specification of the overridden member refines the static specification of the overriding member. Formally,

\[
\mathcal{T} \vdash \{S_d\} \Rightarrow \{S'_d\} \Rightarrow \{S_c\} \Rightarrow \{S'_c\}
\]

**Proof obligation 6.16** Let \(P\) be the program to verify. For any trait \(T\) of \(P\) with superclass \(SC\), the behavioural preservation proof rule 6.15 has to hold for all overriding methods and functions of \(T\).

**Proof obligation 6.17** Let \(P\) be the program to verify. For any class \(D\) of \(P\) with superclass \(C\), the behaviour preservation proof rules 6.15 has to hold for all methods and functions of \(D\).

Observe that the proof obligation for classes is stronger and requires all methods and functions to satisfy the proof rule. The reason is that a class can appear as a superclass in a mixin composition while a trait can not.
6.8.5 Static Superclass Abstraction

We give a proof sketch showing why static superclass abstraction is sound given the proof obligations presented.

**Theorem 6.18** Static superclass abstraction is sound

**Proof sketch** Let $T$ be a trait with declared superclass $SC$.
Let $\text{super.memb}(\text{pars})$ be a late bound super invocation inside of a member of $T$. According to the idea of static superclass abstraction, the specification used to verify the invocation is the static specification $SC_{\text{memb}}$ of the superclass $SC$.

Let $T$ be involved in an arbitrary mixin composition $M$ with linearization order $L_1, L_2, \ldots, L_n, \text{AnyRef}$ where $n \geq 0$ and $L_i$ for $i \in 1, \ldots n$ denotes a trait or class.

Let $t \in 1, \ldots, n$ be such that $L_t = T$ and $sc \in t + 1, \ldots, n$ be such that $L_{sc} = SC$ which has to be uniquely defined by the properties of the linearization order, the requirement that all templates are well-formed and the fact that neither a trait nor a class can extend itself directly or indirectly.

Assume that the super member invocation gets bound to $L_s$ for $s \in t + 1, \ldots, n$ where $L_s$ is such that $L_i$ for $i \in t + 1, \ldots, s - 1$ does not override the member $\text{memb}$. That is, all traits or classes $L_i$ with $i \in t, \ldots, s - 1$ inherit the member $\text{memb}$.

Note: We do not use $L_{t+1}$ as the class or trait to which the super member invocation gets bound as it may only inherit the invoked member. We skip members being inherited and take $L_s$ instead for simplicity.

For static superclass abstraction to be sound, we have to show that the static specification of $\text{memb}$ in $SC$ refines the implementation specification of $\text{memb}$ in $L_s$.

In case $s \geq sc$, the super member invocation gets bound to a member introduced, overridden or inherited by the superclass $SC$. The specification refinement to show follows directly from the static specification conformance proof obligation 6.14, the interface specification conformance proof obligation 6.9 and the fact that $\mathcal{L}(SC) = L_{sc}, \ldots, L_n$ (property of the linearization order).

For the remaining case $s < sc$ we distinguish two cases:

- **Case $L_s$ is a class:**
  Taking into account that $SC \neq L_s$ and using the well-formedness of mixin compositions (definition 6.1), it follows that $L_s$ is a direct or indirect subclass of $SC$. Using the behaviour preservation proof obligation 6.17 for classes, the transitivity of refinements and the static specification conformance proof obligation 6.14, the desired refinement follows.

- **Case $L_s$ is a trait:**
  We have to distinguish two sub-cases. Let $SC_2$ be the superclass of $L_s$. Either the superclass of $L_s$ equals $SC$ (ie. $SC_2 = SC$) or it is another class (ie. $SC_2 \neq SC$) which must be a direct or indirect subclass of $SC$ by the properties of the linearization order.
  In the former case ($SC_2 = SC$) we are done using the behaviour preservation proof obligation 6.16 for traits, the static specification conformance proof obligation 6.14 and the interface specification conformance proof obligation
6.8. DEALING WITH LATE BOUND SUPER INVOCATIONS

6.8.6 Limitation of Static Superclass Abstraction

Our verification strategy uses static superclass abstraction to verify late bound super invocations. That is, we treat every trait in isolation and do not consider them in a concrete mixin composition when verifying a traits implementation.

A Setting we can not Verify

A programmer of a trait $T$ could intend to use $T$ only if another trait $Q$ gets mixed in as well, such that $Q$ is the successor of $T$ in the resulting linearization order of a mixin composition. In this case, super member invocations in $T$ get bound to the implementations in $Q$. Thereby, $T$ could rely on the specification of $Q$ which potentially has a weaker precondition and stronger postcondition than the superclass of $T$.

We are not able to verify such a setting with the presented approach. If such a setting shall be supported without further language constructs, then one needs to follow a per mixin composition body-verification strategy. That is, we verify the implementation of a trait for each mixin composition the trait is involved. As a module may always use a trait from another module, a verification strategy based...
on a per mixin composition is not modular. However, we could still realize it for traits involved in a mixin compositions within the same module. We did not investigate this issue any further.

**Specifications with Requirements**

Investigations in the direction of the before-mentioned limitation should take into account the work of Damiani, Dovland and Schaefer [4]. Paraphrasing their idea in our setting, one could use multiple implementation specifications for members of traits. Each implementation specification may define requirements on the specification of a concrete successor traits or classes in a mixin composition. The requirements would typically be that the successor traits or classes override a certain method of the superclass providing a weaker precondition or a stronger postcondition.

A body then has to satisfy a certain implementation specification only if the specification's requirements are met, whereby the body verification can assume the requirements of the specification in addition.

**Another Solution Approach**

Another solution approach would be to take up the idea of specification referencing as presented by Schwerhoff [25]. Schwerhoff allows the specification of a trait's member to contain references to the pre- and postcondition of its overridden member. Thus, the specification of a member \( m \) of a trait \( T \) is given in terms of the unknown specification of the super class or super trait \( S \) of \( T \).

While we did not investigate how this idea could be applied exactly to our setting, we have the intuition that specification references are a good mean to express the programmers intention of having a trait only used in a mixin composition if the successor trait or class provides the desired guarantees.

**6.9 Automatizing the Proof Obligations**

We outline how the proof obligations presented in this chapter can be automatized and detail the challenges remaining. While the proof obligations can be automatized such that we can verify the rich interface example 6.3.1, the automatation required for the example with the `BackupCell` and `CountingCell` traits 6.3.2 is an open challenge. We discuss the challenge separately in section 6.10.1.

**6.9.1 Behavioural Subtyping and Behaviour Preservation**

The proof obligation for behavioural subtyping 6.8, the proof obligation for behaviour preservation 6.16 of traits and the proof obligation for behaviour preservation 6.17 of classes can readily be automatized in an analogous way as we automatized the proof obligation for behavioural subtyping 4.5 in subclassing-Chalice.
6.9. AUTOMATIZING THE PROOF OBLIGATIONS

6.9.2 Type Specification Conformance

The proof obligation for type specification conformance 6.7 is satisfied by construction. The proof sketch is the analogous to the one given for subclassing-Chalice.

**Theorem 6.19** If the type specifications are generated as described in section 6.6, the type specification conformance proof obligation is satisfied by construction.

**Proof sketch** Let $m$ be an arbitrary member of an arbitrary mixin composition $M$ with linearization $L_1, L_2, \ldots, L_n, \text{AnyRef}$ and associated contract $C$. Let $C := (C^\text{pre}_m, C^\text{post}_m)$ be the code specification of $m$ given in the associated contract $C$ of $M$, let $I_m := (I^\text{pre}_m, I^\text{post}_m)$ be the interface specification of $m$ and let $T_m := (T^\text{pre}_m, T^\text{post}_m)$ be the type specification of $m$. We have to show that $\Gamma \vdash \{I^\text{pre}_m\}\{I^\text{post}_m\} \Rightarrow \{T^\text{pre}_m \land \text{dynamicMixinComposition}(\text{this}) = M\}\{T^\text{post}_m\}$ holds.

From the interface specification transformation given in section 6.6.2 we know the following definition.

1. If $m$ is an external member, then all unspecified indices are dynamic.
   That is $I_m := \mathcal{I}(C_m, \text{dynamic})$

2. If $m$ is an internal member, then use the syntactical indexing transformation.
   That is $I_m := SI(C_m, L_1)$

Furthermore, from the type specification transformation given in section 6.6.3 we know the following definition.

$$T_m := \mathcal{I}(C_m, \text{dynamic})$$

Using $T^\text{pre}_m \land \text{dynamicMixinComposition}(\text{this}) = M$ we can re-index all abstract families associated with this and indexed by dynamic with the new index $T_1$ as this is the trait or class appearing first in the linearization.

We can do this obviously for the type specification and for less obvious reasons as well for the interface specification. The later can be argued as we did for the proof sketch for the type conformance proof obligation in section 5.5.1.

For the re-indexed interface specification $I'$ we have

1. If $m$ is an external member we have
   $$I'_m := SR(\mathcal{I}(C_m, \text{dynamic}), L_1, \text{dynamic}) = SR(SI(C_m, L_1), L_1, \text{dynamic})$$

2. If $m$ is an internal member we have
   $$I'_m := SR(SI(C_m, L_1), L_1, \text{dynamic})$$

For the re-index type specification $T'$ we have

$$T'_m := SR(\mathcal{I}(C_m, \text{dynamic}), L_1, \text{dynamic}) = SR(SI(C_m, L_1), L_1, \text{dynamic})$$

It follows readily that the two specifications are syntactically equal. \(\square\)

---

2This is analogous to the case in subclassing-Chalice where the dynamic class corresponds as well to the first class in the linearization
6.9.3 Static Specification Conformance

The static specification conformance proof obligation 6.14 is satisfied by construction in an analogous way to the type conformance proof obligation.

**Theorem 6.20** If the static specifications are generated as described in section 6.8.3, the static specification conformance proof obligation 6.14 is satisfied by construction.

**Proof sketch** We give the proof sketch for classes only. The proof sketch for traits is analogous. Let \( m \) be an arbitrary member of an arbitrary class \( D \) with associated contract \( C \). Let \( C_m := (C_m^{\text{pre}}, C_m^{\text{post}}) \) be the code specification of \( m \) given in the associated contract \( C \) of the class, let \( I_m := (I_m^{\text{pre}}, I_m^{\text{post}}) \) be the interface specification of \( m \) and let \( S_m := (S_m^{\text{pre}}, S_m^{\text{post}}) \) be the static specification of \( m \). We have to show that \( \Gamma \vdash \{ I_m^{\text{pre}} \} \{ I_m^{\text{post}} \} \Rightarrow \{ S_m^{\text{pre}} \land \text{this} : D \} \{ S_m^{\text{post}} \} \) holds.

From the interface specification transformation given in section 6.6.2 we know the following definitions. We directly use the fact that \( D \) is the first element of its linearization order.

1. If \( m \) is an external member, then all unspecified indices are dynamic.
   That is \( I_m := I(C_m, \text{dynamic}) \)

2. If \( m \) is an internal member, then use the syntactical indexing transformation.
   That is \( I_m := SI(C_m, D) \)

Furthermore, from the static specification transformation for classes given in section 6.8.3 we know the following definition.

1. If \( m \) is an external member, then all unspecified indices are dynamic.
   That is \( S_m := I(C_m, \text{dynamic}) \)

2. If \( m \) is an internal member, then use the syntactical indexing transformation.
   That is \( S_m := SI(C_m, \text{static}) \)

Using \( \text{this} : D \) we can re-index all abstract families associated with \( \text{this} \) and indexed by \( \text{static} \) with the new index \( D \). That is, we replace the static symbol with its concrete value. In the following we make use of the fact that the code specifications are not allowed to contain symbols such as \( \text{static} \) to be given explicitly by the programmer. As a consequence, we do not need to re-index the interface specification as it can not contain a static symbol.

For the re-index static specification \( S' \) we have the following. We make use of the fact that \( C_m \) may not contain a static symbol.

1. If \( m \) is an external member we have
   That is \( S'_m := SR(I(C_m, \text{dynamic}), D, \text{static}) = I(C_m, \text{dynamic}) \)

2. If \( m \) is an internal member we have
   That is \( S'_m := SR(SI(C_m, \text{static}), D, \text{static}) = SI(C_m, D) \)

It follows readily that the two specifications are syntactically equal. \( \square \)
6.10. OPEN CHALLENGES

6.9.4 Interface Conformance

The proof obligation for interface conformance 6.9 is the hardest to automate. The approach used in subclassing-Chalice and described in section 5.5.2 can as well be used and is expected to allow verification of the rich interface example given in section 6.3.1.

However, the approach will fail for the example 6.3.2. The problem is that the interface specification of a method specifies the effect of all traits mixed in. On the other hand, the implementation specification used is from the last trait overriding the method. We detail this in the following section 6.10.1

6.10 Open Challenges

This section identifies open challenges to the verification of programs involving traits. We are strongly convinced that they all can be solved with the introduction of a new specification construct. We do not provide this new specification construct but refer to the idea of specification referencing presented by Schwerhoff in [25]. Shortly summarizing the idea of specification referencing, specifications of members may reference the pre- and post-conditions of the overridden method or function even if the overridden member is statically not known. At mixin composition time, the referenced specifications get injected and checked without reverifying the body.

6.10.1 Verifying Traits Introducing New Side-Effects

We first outline the need for a new specification feature and then show how the lack of it prevents us to verify the running example 6.3.2 with the traits BackupKeeper and InvocationCounter.

The Super Call Specification Challenge

Traits may override methods of the superclass in order to trigger effects on their own state or to change the implementation as long as it remains within the specified behaviour.

Our running example with traits BackupKeeper and InvocationCounter given in listing 6.5 provides classes where two traits override the set method of the superclass. Both overriding methods of the traits have a postcondition specifying new effects on the state added by their traits. Note that both overriding methods of the traits use a super call in order to preserve the effect of the successor trait in a mixin composition.

Both classes CountingBackupCell and BackupCountingCell respect the contract BackupingAndCountingCellContract which is given in listing 6.4. Thereby, the two classes clearly express the intention of the programmer to preserve the effect of
both traits being mixed in. In particular, this example demonstrates a case where the traits shall behave symmetrically.

Let us restrict to the class CountingBackupCell. From subclassing-Chalice we are used to that within a linearization order of a class, the interface specification of an overriding member includes the effect of the overridden member. This is an indirect consequence of the behavioural subtyping proof obligation for subclassing-Chalice. However, for traits this is not the case any more. In CountingBackupCell, the trait BackupKeeper overrides the set method of InvocationCounter but the specification of BackupKeeper does not specify the effect of the overridden method of InvocationCounter. This is despite of both implementations invoking the super method which preserves the behaviour of the overridden method. The reason for BackupKeeper not to specify the effect of its successor trait is clearly that the effect to be specified is not known statically.

We observe the following discrepancy. While a trait can preserve the unknown effect of an overridden method using a late bound super call, he can not specify that he preserves the unknown effect. What we need in order to resolve this discrepancy is a mean to allow specifications to express that a method invocation preserves the unknown behaviour of the overridden method. We denote this need for a new specification feature the *super call specification challenge*.

**Automating the Interface Verification and Its Challenges**

Still looking at class CountingBackupCell, we can provide the interface specification and implementation specification of the class.

```plaintext
Implementation specification of mixin composition CountingBackupCell {
var val: int
var bak: int
var counter: int

predicate valid { super.valid && acc(bak) }
// Inherited from BackupKeeper
```
6.10. OPEN CHALLENGES

Listing 6.12: Implementation specification of class CountingBackupCell. Observe that the set method does not specify the effect of the trait InvocationCounter.

Listing 6.13: Interface specification of class CountingBackupCell. Observe that the set method specifies the effects of traits BackupKeeper and InvocationCounter.

Looking at the set method, we observe that the interface specification includes the effect of both traits BackupKeeper and InvocationCounter, whereas the implementation specification only includes the effect of the BackupKeeper.

We can either see this as a problem of our implementation specification transformation given in 6.8.3 or we can consider it as an instance of the super call specification challenge. However, adapting the implementation specification generation
will as well raise the question whether an overriding member in a trait preserves the effect of a call to the overridden member or whether the specified behaviour has only been re-implemented. We give an example

The Consequence of Not Using a Super Call

```scala
trait NoSuperCall extends BasicCell {
  var newVal : int

  override predicate valid { super.valid & acc(newVal) }

  override function get(): int
    requires valid
    { unfolding valid in newVal }

  override method set(v: int)
    requires valid
    ensures valid & get() == v
    {
      unfold valid<NoSuperCall>
      newVal := v
      fold valid<NoSuperCall>
    }
}
```

Listing 6.14: Trait reimplementing the behaviour of BasicCell but not invoking the super method.

```scala
class BrokenClass
  extends BasicCell
  with InvocationCounter with NoSuperCall with BackupKeeper
  respects BackupingAndCountingCell {
    // no definitions given
  }

  // Linearization:
  // BrokenClass, BackupKeeper, NoSuperCall, ChangeCounter, BasicCell, AnyRef

  contract InvocationCountingCellContract refines BasicCellContract {
    method set(v: int)
      requires valid
      ensures valid
      ensures getCounter() = old(getCounter()) + 1
    }

    function getCounter(): int
      requires valid
    }

    function get(): int
      requires valid
    }

  // Note: A contract without a body merely establishes
  // the refinement relation between the two contracts
  // that are already defined elsewhere.

  contract BackupingAndCountingCellContract refines InvocationCountingCellContract

  var o : BasicCell with InvocationCounter respects InvocationCountingCellContract
```
6.10. OPEN CHALLENGES

Listing 6.15: Mixin composition involving the BackupKeeper, InvocationCounter and NoSuperCall traits. We provide as well the contract InvocationCountingCell describing one potential supertype of BrokenClass. The BackupingAndCountingCellContract contract is given in listing B.18. This is an example that correctly fails verification.

To see the problem assume we have a non-null reference o with static type Cell with InvocationCounter. The reference my contain an instance of BrokenClass. However, invoking the set will not provide the behaviour of InvocationCounter. The reason is that the method set of the trait InvocationCounter is never executed. This comes from the implementation of trait NoSuperCall which does not call the super method and is executed before the implementation of InvocationCounter.

Summary of Challenge

In order to resolve the super call specification challenge, an additional language construct for specifications seems to be necessary. While this open challenge prevents the verification in some cases (i.e the example discussed in section 6.3.2), our proposed verification approach is sound and can verify trait-Chalice programs as long as all traits within the mixin compositions override different members or only override members without changing the specification of the overridden member.

6.10.2 Specifying Constructors

Another open challenge comes with the specification of constructors. We have omitted constructors in the examples which means that a default constructor gets generated. The default constructor return the bare access permissions and it is the clients task to fold the required predicate properly.

However, we would like to have constructors providing already the folded predicates as we did for subclassing-Chalice. This means that a constructor specifies in its postcondition the returned predicates. The challenge is, that a constructor of a trait does not know the specification of its late bound super constructor call as this depends on a concrete linearization order induced by a mixin composition. Thus, there is currently no way a constructor of a trait can correctly specify the returned predicates and access permissions. We would like to point out that while we discussed the challenge using the access permissions, it readily applies to all sort of guarantees a constructor gives in its postcondition.

On possible solution is to let constructors reference the postcondition of their super class’ or trait’s constructor. The referencing of the postcondition could be done based on ideas of Schwerhoff. Without a solution to this challenge, traits can still be used. However, it forces the programmer to fold all predicates manually after the instantiation of a new mixin composition.
We successfully transferred the work of Parkinson and Bierman [19, 20] from the separation logic context to the implicit dynamic frame context and used it to add support for behavioural subclassing to Chalice and Syxc.

On the theoretical level, we have introduced the concepts of implementation, interface and type specifications allowing a more general treatment of the theory and the separation of behavioural subtyping and inheritance when it comes to verification. Furthermore, we adapted the idea of abstract predicate families to abstract function families.

With respect to Chalice, we have extended the language with support for subclassing (inheritance and subtyping), overriding and respecification of methods and functions, overriding of predicates, constructors and have provided a new semantics for monitor invariants. Furthermore, we have taken care of credit expressions with respect to specification refinements and have adapted the fork and join statements.

As a last step, we have outlined how support for traits can be added to Chalice and Syxc based on the developed theory.

Reviewing our goals from section 1.1, we can conclude that we successfully added support for subclassing to Chalice and that we were able to add verification support to Syxc without giving up existing language features. Furthermore, we outlined how verification support for traits can be realized but did not yet provide an implementation.

7.1 Related Work

Support for subclassing can be found in several automated program verifiers. Distefano and Parkinson have developed jStar [6] targeting Java and Jacobs and Piessens developed VeriFast [7] targeting Java and C. Both use symbolic execution for the verification. Symbolic execution and verification condition generation at the same time is supported by VeriCool [27] developed by Smans, Jacobs and Piessens. VeriCool is the one closest to our work as it is based on implicit dynamic frames approach as well.
Up to our knowledge, traits have so far not found their way into program verifiers. Schwerhoff [25] as well as Damiani, Dovland and Schaefer [4] have investigated traits on a theoretical level, where the later plan to implement their proposed proof system in KeY [2].

7.2 Outlook

We present ideas for the further development of Chalice and Syxc or simply point out problems not being solved satisfactorily so far.

7.2.1 Heuristics for Consume

We have discussed the idea of a consume heuristics in section 4.4.5 and explained why we decided against implementing one so far.

Without a consume heuristics, we can not check specification refinements without tailor-made symbolic execution rules. While we gave tailor-made rules working for the general case, it currently limits the use of explicit indices in abstract predicate families appearing in specifications (see section 5.5.4). As we expect a consume heuristics to resolve this limitation and as it would allow to simplify the symbolic execution rules, we propose to investigate the feasibility of such a consume heuristics in more detail.

In order to make the consume heuristics simpler, one could consider taking up the idea of inheriting predicate bodies instead of aggregating predicate bodies when inheriting predicates (see section 5.4). As it seems, a consume heuristics for the notion of inheriting predicate bodies seems to be much simpler to realize.

7.2.2 Auto Respecification of Inherited Members

In section 3.5.7 we have motivated respecification of members. In particular, we have detailed why it will often be necessary to respecify inherited methods in order to guarantee the absence of effect on added state in the inheriting class.

The respecification often duplicates the entire specification of the inherited member only to add a few more constraints to the postcondition. Furthermore, the added constraints do in most cases only express that the method does not change the returned value of some function introduced in the inheriting class. Thus, respecification for inherited members seems to be a repetitive task, code-duplicating and promisingly to automate.

We propose therefore to investigate how the respecification of inherited members could be automated and how it would have to be reflected in the programming language Chalice. A minimal solution consists of providing a mean to minimize specification-duplication for the purpose of respecification. This could be by allowing a specification to reference the specification of the super class in some way. The optimal solution would be that the respecification works fully automatically, in particular for examples as the one given in section 3.5.7.
7.2. OUTLOOK

7.2.3 Statically and Dynamically Dispatched Invocations

Statically bound and dynamically bound member invocations on this within the same method is not possible in the current implementation. We discussed this in section 3.5.11. For internal methods we explained that dynamically bound function applications are not possible in general due to potentially missing access permissions. As it seems, it is often the case in practice that functions having the role of a getter could be invoked safely as they are not going to change in subclasses and therefore do not need additional access permissions.

The following listing is motivated by an example of Parkinson and Bierman [20]. The challenge is that subclasses may override the get function in an incompatible way.

```java
0  class SubCell extends Cell {
1     override predicate valid { this.valid<Cell> }
2
3     override method doNothing() {
4         requires valid
5         ensures valid
6         ensures get() == old(get())
7     }
8
9     var v := this.get() // dynamically bound call
10     call super.set(v)    // statically bound call
11     // Body verification (in the current implementation)
12     // fails correctly on the function application to 'get'
13 }```

Listing 7.1: SubCell class with special implementation of the set method. The Cell class is as given in listing A.1.

We propose to investigate how statically and dynamically bound member invocations within the same method or function could be supported. As a starting point one should investigate the effect of a ‘final’ modifier for methods and functions in order to understand the additional guarantees arising from this. We then propose to take up the idea of Parkinson and Bierman [20], namely to impose additional restrictions on subclasses.

7.2.4 Interfaces in the Sense of Java

In order to gain more experience with abstract predicate families, we propose to add interfaces as known from Java to Chalice. While it does not align with the long-term goal of supporting the Scala language directly, interfaces can be seen as restricted usage of traits and would already allow to investigate interesting aspects.

An interesting question, for instance, is whether interfaces favour the notion of predicate inheritance by aggregation or by predicate body inheritance.
7.2.5 Traits

We outlined in chapter 6 the general idea of how traits could be supported using the presented formalisms. We propose to further investigate this topic and to provide an implementation.

In particular we propose to take up the idea of *abstract specifications* as presented in [25]. An abstract specification allows references to the superclass in specifications. It allows, for instance, to express that a postcondition of a called method still holds at the end of the executing method. In connection with traits and behavioural subtyping this seems to be the key for a flexible use of traits. In particular, we expect it to be a potential solution to the open challenges identified for traits in section 6.10.
This section contains the detailed examples for which was not enough space at the place of discussion.

### A.1 Cell and Recell Example

Cell and Recell example inspired by an example of Parkinson and Bierman in [20]. We provide as well its implementation, static and type specification.

```plaintext
class Cell {
    var val: int

    predicate valid { acc(val) }

    primary this ()
    ensures valid && get() == 0
    {
        call super()
        val := 0
        fold valid<Cell>
    }

    this(v: int)
    ensures valid && get() == v
    {
        call this()
        call set(v)
    }

    function get(): int
    requires valid
    ensures true
    { unfolding valid<Cell> in val }

    method set(v: int)
    requires valid
    ensures valid
    ensures get() == v
```
APPENDIX A. EXAMPLES FOR SUBCLASSING

Listing A.1: Cell class

```java
class Cell {
    var val : i
    method dynamic add(v : int)
        requires valid
        ensures valid
        ensures get() == old(get()) + v
        {
            var currentVal := get()
            call set(currentVal + v)
        }
    }
}

class Recell extends Cell {
    var bak : int
    override predicate valid { this.valid<Cell> && acc(bak) }
    primary this() 
        ensures valid && get() == 0 && getBak() == 0
    {
        call super()
        bak := 0
        fold valid<Recell>
    }
    this(v : int)
        ensures valid && get() == v && getBak() == 0
    {
        call this()
        call set(v)
    }
    function getBak(): int
        requires valid
        ensures true
        { unfolding valid<Recell> in bak }
    override method set(v: int)
        requires valid
        ensures valid
        ensures get() == v
        ensures getBak() == old(get())
        {
```
A.1. CELL AND RECELL EXAMPLE

A.1.1 Implementation Specifications

Implementation specification of class Cell {
  var val: int
  predicate valid { acc(val) }
  function get(): int
    requires this.valid<Cell>
    ensures true
  method set(v: int)
    requires this.valid<Cell>
    ensures this.valid<Cell>
    ensures this.get<Cell>() == v
  method add(v: int)
    requires this.valid<dynamic>
    ensures this.valid<dynamic>
    ensures this.get<dynamic>() == old(this.get<dynamic>()) + v
  method addAlternative(v: int)
    requires this.valid<Cell>
    ensures this.valid<Cell>
    ensures this.get<Cell>() == old(this.get<Cell>()) + v
}

Listing A.3: Implementation specification of class Cell.
APPENDIX A. EXAMPLES FOR SUBCLASSING

A.1.2 Interface Specifications

```
| Interface specification of class Cell { |
| var val: int |
| predicate valid { acc(val) } |
| function get(): int |
| requires this.valid<Cell> |
| ensures true |
| method set(v: int) |
| requires this.valid<Cell> |
| ensures this.valid<Cell> |
| ensures this.get<Cell>() == v |
| method add(v: int) |
| requires this.valid<dynamic> |
| ensures this.valid<dynamic> |
| ensures this.get<dynamic>() == old(this.get<dynamic>()) + v |
| method addAlternative(v: int) |
| requires this.valid<Cell> |
| ensures this.valid<Cell> |
| ensures this.get<Cell>() == old(this.get<Cell>()) + v |
```

Listing A.5: Interface specification of class Cell

```
| Interface specification of class Recell { |
| var bak: int |
| predicate valid { this.valid<Cell> && acc(bak) } |
| function getBak(): int |
| requires this.valid<Recell> |
| ensures true |
| method set(v: int) |
| requires this.valid<Recell> |
| ensures this.valid<Recell> |
```

Listing A.4: Implementation specification of class Recell

```
// Inherited from Cell
method add(v: int)
  requires this.valid<dynamic>
  ensures this.valid<dynamic>
  ensures this.get<dynamic>() == old(this.get<dynamic>()) + v

// Inherited from Cell
method addAlternative(v: int)
  requires this.valid<Cell>
  ensures this.valid<Cell>
  ensures this.get<Cell>() == old(this.get<Cell>()) + v
```
A.1. CELL AND RECELL EXAMPLE

ensures \( \text{this}. \text{getReCell}() == v \)
ensures \( \text{this}. \text{getBakReCell}() == \text{old}(\text{this}. \text{getReCell}()) \)

function get(): int
  requires \( \text{this}. \text{validReCell} \)
  ensures true

method add(v: int)
  requires \( \text{this}. \text{valid<dynamic>} \)
  ensures \( \text{this}. \text{valid<dynamic>} \)
  ensures \( \text{this}. \text{get<dynamic>}() == \text{old}(\text{this}. \text{get<dynamic>}) + v \)

method addAlternative(v: int)
  requires \( \text{this}. \text{valid<ReCell>} \)
  ensures \( \text{this}. \text{valid<ReCell>} \)
  ensures \( \text{this}. \text{get<ReCell>}() == \text{old}(\text{this}. \text{get<ReCell>}) + v \)

Listing A.6: Interface specification of class Recell

A.1.3 Type Specifications

Type specification of class Cell {
  \textit{var} val: int

  \textit{predicate} valid

  \textit{function} get(): int
    \textit{requires} \( \text{this}. \text{valid<dynamic>} \)
    \textit{ensures} true

  \textit{method} add(v: int)
    \textit{requires} \( \text{this}. \text{valid<dynamic>} \)
    \textit{ensures} \( \text{this}. \text{valid<dynamic>} \)
    \textit{ensures} \( \text{this}. \text{get<dynamic>}() == \text{old}(\text{this}. \text{get<dynamic>}) + v \)

  \textit{method} addAlternative(v: int)
    \textit{requires} \( \text{this}. \text{valid<dynamic>} \)
    \textit{ensures} \( \text{this}. \text{valid<dynamic>} \)
    \textit{ensures} \( \text{this}. \text{get<dynamic>}() == \text{old}(\text{this}. \text{get<dynamic>}) + v \)
}

Listing A.7: Type specification of class Cell
function getBak(): int
    requires this.valid<dynamic>
    ensures true

method set(v: int)
    requires this.valid<dynamic>
    ensures this.valid<dynamic>
    ensures this.get<dynamic>() == v
    ensures this.getBak<dynamic>() == old(this.get<dynamic>())

method add(v: int)
    requires this.valid<dynamic>
    ensures this.valid<dynamic>
    ensures this.get<dynamic>() == old(this.get<dynamic>()) + v

method addAlternative(v: int)
    requires this.valid<dynamic>
    ensures this.valid<dynamic>
    ensures this.get<dynamic>() == old(this.get<dynamic>()) + v
}

Listing A.8: Type specification of class Recell
### Examples for Traits

#### B.1 BasicCell class

```java
class BasicCell {
    var val: int

    predicate valid { acc(val) }

    primary this() {
        ensures valid && get() == 0
        { call super();
          val := 0
          fold valid<BasicCell>
        }
    }

    this(v: int) {
        ensures valid && get() == v
        { call this();
          call set(v)
        }
    }

    function get(): int
        requires valid
        ensures true
        { unfolding valid<BasicCell> in val }

    method set(v: int)
        requires valid
        ensures valid
        ensures get() == v
        { unfold valid<BasicCell>
          val := v
          fold valid<BasicCell>
        }
}
```

*Listing B.1: BasicCell class.*
B.1.1 Contract

```plaintext
class BasicCellContract {
  var val: int
  predicate valid
  function get(): int requires valid
  method set(v: int) requires valid ensures valid ensures get() == v
}
```

Listing B.2: Contract for the BasicCell class.

B.1.2 Implementation Specifications

```plaintext
impl BasicCell {
  var val: int
  predicate valid { acc(val) }
  primary this () {
    ensures valid<BasicCell>
    ensures get<BasicCell>() == 0
  }
  this(v: int)
    ensures valid<BasicCell>
    ensures get<BasicCell>() == v
  function get(): int requires valid<BasicCell>
  method set(v: int) requires valid<BasicCell>
    ensures valid<BasicCell>
    ensures get<BasicCell>() == v
}
```

Listing B.3: Implementation specification of trait BasicCell.

B.1.3 Static Specifications

```plaintext
Static specification of mixin composition BasicCell {
  var val: int
  predicate valid
  function get(): int requires valid<static>
  method set(v: int)
```
B.2. RICH INTERFACES

Listing B.4: Static specification of the class BasicCell.

B.1.4 Interface Specifications

Listing B.5: Interface specification of trait BasicCell.

B.1.5 Type Specifications

Listing B.6: Type specification of trait BasicCell.

B.2 Rich Interfaces
trait Swapping extends BasicCell {
    method dynamic swap(other: BasicCell)
        requires valid
        requires other != null & other.valid
        ensures valid
        ensures other.valid
        ensures this.get() == old(other.get())
        ensures other.get() == old(this.get())
    {
        var tmp := other.get()
        call other.set(get())
        call set(tmp)
    }
}

Listing B.7: Trait Swapping enriching the interface of class BasicCell. The BasicCell class is as given in listing B.1.

trait Adding extends BasicCell {
    method dynamic add(v: int)
        requires valid
        ensures valid
        ensures old(get()) + v == get()
    {
        var newVal := get() + v
        call set(newVal)
    }
}

Listing B.8: Trait Adding enriching the interface of class BasicCell. The BasicCell class is as given in listing B.1.

class AddingAndSwappingCell extends BasicCell
    with Adding with Swapping
    respects AddingAndSwappingCellContract
    {
        // All members inherited
    }

Listing B.9: Mixin composition involving the Swapping and Adding traits.

B.2.1 Contract

contract AddingAndSwappingCellContract refines BasicCellContract {
    var val: int

    predicate valid

    method set(v: int)
        requires valid
        ensures valid
        ensures get() == v

    function get(): int
        requires valid

    function add(v: int)
        requires valid
        ensures old(get()) + v == get()
}

Listing B.10: Contract AddingAndSwappingCellContract refines BasicCellContract.
B.2. RICH INTERFACES

13
14
15
16
17
18
19
20
21
22
23
24
25

method dynamic add(v: int)
requires valid
ensures valid
ensures old(get()) + v == get()

method dynamic swap(other: BasicCell)
requires valid
requires other != null && other.valid
ensures valid
ensures other.valid
ensures this.get() == old(other.get())
ensures other.get() == old(this.get())
}

Listing B.10: Contract used by AddingAndSwappingCell. The BasicCellContract is given in listing B.2.

B.2.2 Implementation Specifications

Implementation specification of trait Swapping {

method swap(other: BasicCell)
requires this.valid<dynamic>
requires other != null && other.valid<dynamic>
ensures this.valid<dynamic>
ensures other.valid<dynamic>
ensures this.get<dynamic>() == old(other.get<dynamic>())
ensures other.get<dynamic>() == old(this.get<dynamic>())
}

Listing B.11: Implementation specification of trait Swapping.

Implementation specification of mixin composition AddingAndSwappingCell {

var val: int

predicate valid

method set(v: int)
requires valid<BasicCell>
ensures valid<BasicCell>
ensures get<BasicCell>() == v

function get(): int
requires valid<BasicCell>

method add(v: int)
requires valid<dynamic>
ensures valid<dynamic>
ensures old(get<dynamic>()) + v == get<dynamic>()

method swap(other: BasicCell)
requires valid<dynamic>
requires other != null && other.valid<dynamic>
ensures valid<dynamic>
ensures other.valid<dynamic>
ensures this.get<dynamic>() == old(other.get<dynamic>())
ensures other.get<dynamic>() == old(this.get<dynamic>())
}

### B.2.3 Interface Specifications

**Listing B.12:** Implementation specification of mixin composition `AddingAndSwappingCell`.

```java
Listing B.12: Implementation specification of mixin composition AddingAndSwappingCell.

```java

```java
Interface specification of mixin composition AddingAndSwappingCell {
  var val : int

  predicate valid

  method set(v: int)
    requires valid<AddingAndSwappingCell>
    ensures valid<AddingAndSwappingCell>
    ensures get<AddingAndSwappingCell>() == v

  function get(): int
    requires valid<AddingAndSwappingCell>

  method add(v: int)
    requires valid<dynamic>
    ensures valid<dynamic>
    ensures old(get<dynamic>()) + v == get<dynamic>()

  method swap(other: BasicCell)
    requires valid<dynamic>
    requires other != null && other.valid<dynamic>
    ensures valid<dynamic>
    ensures other.valid<dynamic>
    ensures this.get<dynamic>() == old(other.get<dynamic>())
    ensures other.get<dynamic>() == old(this.get<dynamic>())
}
```

**Listing B.13:** Interface specification of mixin composition `AddingAndSwappingCell`.

### B.2.4 Type Specifications

**Listing B.14:** Type specification of mixin composition `AddingAndSwappingCell`.

```java
Listing B.14: Type specification of mixin composition AddingAndSwappingCell.

```java

```java
Type specification of mixin composition AddingAndSwappingCell {
  var val : int

  predicate valid

  method set(v: int)
    requires valid<dynamic>
    ensures valid<dynamic>
    ensures get<dynamic>() == v

  function get(): int
    requires valid<dynamic>

  method add(v: int)
    requires valid<dynamic>
    ensures valid<dynamic>
    ensures old(get<dynamic>()) + v == get<dynamic>()
```
B.3 Traits Adding Side-Effects

```scala
trait BackupKeeper extends BasicCell {
  var bak: int

  override predicate valid { super.valid & & acc(bak) }

  function getBak(): int
    requires valid
  {
    unfolding valid in bak
  }

  override method set(v: int)
    requires valid
    ensures valid
    ensures get() == v
    ensures getBak() == old(get())
    {
      unfold valid<BackupKeeper>
      var bak := super.get()
      call super.set(v)
      fold valid<BackupKeeper>
    }
}
```

Listing B.15: Trait example with late bound super call. The BasicCell class is as given in listing B.1.

```scala
trait InvocationCounter extends BasicCell {
  var counter: int

  override predicate valid { super.valid & & acc(counter) }

  function getCounter(): int
    requires valid
  {
    unfolding valid in counter
  }

  override method set(v: int)
    requires valid
    ensures valid
    ensures get() == v
    ensures getCounter() == old(getCounter()) + 1
    {
      unfold valid<InvocationCounter>
      counter := counter + 1
      call super.set(v)
      fold valid<InvocationCounter>
    }
}
```

Listing B.16: Trait example with late bound super call. The BasicCell class is as given in listing B.1.
APPENDIX B. EXAMPLES FOR TRAITS

Listing B.16: Trait example with late bound super call. The BasicCell class is as given in listing B.1.

class CountingBackupCell
  extends BasicCell with InvocationCounter with BackupKeeper with BackUpingAndCountingCellContract {
  // no definitions given
}
// Linearization:
// CountingBackupCell, BackupKeeper, InvocationCounter, BasicCell, AnyRef

class BackupCountingCell
  extends Cell with BackupKeeper with InvocationCounter with BackUpingAndCountingCellContract {
  // no definitions given
}
// Linearization:
// CountingBackupCell, InvocationCounter, BackupKeeper, BasicCell, AnyRef

Listing B.17: Mixin composition involving the BackupKeeper and InvocationCounter traits. The class names reflect the order in which traits appear in the template.

B.3.1 Contract

contract BackUpingAndCountingCellContract refines BasicCellContract {
  var val: int
  var bak: int
  var counter: int

  predicate valid

  method set(v: int)
    requires valid
    ensures valid
    ensures get() == v
    ensures getCounter() == old(getCounter()) + 1
    ensures getBak() == old(get())

  function getCounter(): int
    requires valid

  function getBak(): int
    requires valid

  function get(): int
    requires valid
}

Listing B.18: Contract used by BackupCountingCell and CountingBackupCell in listing 6.5. Observe in particular the postcondition of the set method combining the postconditions of the traits BackupKeeper and InvocationCounter. The BasicCellContract contract is given in listing B.2.
B.3.2 Implementation Specifications

APPENDIX B. EXAMPLES FOR TRAITS


B.3.3 Static Specifications

Listing B.21: Static specification of the trait BackupKeeper.

Listing B.22: Static specification of the trait InvocationCounter.

B.3.4 Interface Specifications

B.3. TRAITS ADDING SIDE-EFFECTS

```
12 == old (getCounter<CountingBackupCell>()) + 1
13 ensures getBak<CountingBackupCell>()
14 == old (get<CountingBackupCell>())
15
function getCounter (): int
16 requires valid<CountingBackupCell>
17
function getBak (): int
18 requires valid<CountingBackupCell>
19
function get (): int
20 requires valid<CountingBackupCell>

```

```
0 Interface specification of mixin composition BackupCountingCell {
1 var val: int
2 var bak: int
3 var counter: int
4
5 predicate valid
6
7 method set(v: int)
8 requires valid<BackupCountingCell>
9 ensures get<BackupCountingCell>() == v
10 ensures getCounter<BackupCountingCell>()
11 == old (getCounter<BackupCountingCell>()) + 1
12 ensures getBak<BackupCountingCell>()
13 == old (get<BackupCountingCell>())
14
function getCounter (): int
15 requires valid<BackupCountingCell>
16
function getBak (): int
17 requires valid<BackupCountingCell>
18
function get (): int
19 requires valid<BackupCountingCell>

```

B.3.5 Type Specifications

```
0 Type specification of mixin composition CountingBackupCell {
1 var val: int
2 var bak: int
3 var counter: int
4
5 predicate valid
6
7 method set(v: int)
8 requires valid<dynamic>
9 ensures valid<dynamic>
10 ensures get<dynamic>() == v

```
Listing B.25: Type specification of mixin composition CountingBackupCell.

Listing B.26: Type specification of mixin composition BackupCountingCell.
Bibliography


