ETH ZÜRICH

MASTERS THESIS

Automating Modular Reasoning About Higher-Order Functions

Author:
Benjamin WEBER

Supervisors:
Prof. Dr. Peter MÜLLER
Arshavir TER-GABRIELYAN
Marco EILERS

Chair of Programming Methodology
Department of Computer Science

November 12, 2017

ETH ZÜRICH

Abstract

Chair of Programming Methodology
Department of Computer Science

Master of Science

Automating Modular Reasoning About Higher-Order Functions

by Benjamin WEBER

Automatic modular verification is a user-friendly and scalable solution to gain strong guarantees about software correctness. Higher-order functions and closures are attractive features of a programming language that allow for flexible and extensible code. However, verification thereof poses significant challenges since contracts of closures are generally unknown. In this thesis we propose a methodology that consists of so called *call slots* and *parametric assertions* which we partially implemented. We find that we can verify a wide range of interesting use cases with our methodology and that it is suitable for automatic modular verification. This gives us strong support for closures and higher-order functions in a formal verification setting.

Contents

List of Figures

Chapter 1

Introduction

With the increasing prevalence of software in everyday life, software bugs become more costly. To alleviate this, strong means to ensure software correctness are required. While software testing is a proven approach for software correctness, it gives no guarantees that a software is free of errors. In contrast formal verification guarantees that a software adheres to its specifications. For practical purposes, it is desirable to automate as much of formal verification as possible. This can be achieved by encoding the verification problem into first-order logic so that efficient automatic SMT solvers can be used. With the increasing size of software, scalability of verification also becomes a concern. Modular verification scales well and allows to verify code that uses libraries where only the specifications of the libraries are available. Research in this are has seen much progress in the last decade. As a result there exist now tools for popular programming languages such as Java and C that are modular and can automatically verify implementations w.r.t. to their specifications.

1.1 Closures

Many programming languages allow one to assign methods to variables. Such variables are called *closures*. In many cases it is unknow to which method a closure points.

Often programming languages with nested methods allow methods to *capture* variables from outer methods. This means nested methods can read and modify variables declared in outer methods.

Higher-order functions are functions that take closures as parameters. Calling such a parameter inside a function can lead to any code being executed, because a client of a higher-order function can use any function for that parameter. This makes higher-order functions very customizable and extensible since a client can create new methods for this parameter depending on their needs.

1.2 Viper and Nagini

Viper [23] is an automatic deductive modular verification infrastructure. It features an intermediate verification language called the Viper language. Front-ends to Viper can then translate other programming languages to the Viper language.

Nagini is a front-end that translates Python programs with static type annotations to Viper's intermediate language. Furthermore, Nagini expects programs to be annotated with verification constructs such as preconditions, postconditions and loop invariants.

1.3 Problem Statement

While closures are an attractive feature for programmers, they complicate verification of code. Since we may not modularly know to which method a closure points, we cannot always know its contracts. The questions arise: How can we justify that it is ok to call a closure at a certain point and how do we know what holds true after a closure call?

In this thesis we want to explore approaches to this problem and develop a methodology that allows us to verify Python closures in Nagini. In particular, we are looking for approaches that allow for modular verification and are suitable for automation by SMT solvers. However, we limit ourselves to closures without capturing.

Chapter 2

State of the Art

For verifying higher-order function, many approaches have been proposed. These approaches can differ significantly depending on their context. E.g., there exist many approaches that translate the problem into a higher-order logic problem [11, 12, 4, 27, 24, 18, 13]. Viper [23] is deliberately limited to first-order logic so that efficient SMT solvers such as z3 [8] can be used.

Another interesting approach is to use so called *model programs* [26]. They are based on a *greybox* approach [6]. Viper uses a blackbox approach where a method is specified by a precondition and a two-state postcondition. A verifier can only use these specifications for verification. In the greybox approach this limitation is weakened so that in addition to preconditions and postconditions one can specify, e.g., that a method carries out certain calls. With model programs, these calls can be closure calls. Using this approach it is possible to verify whether a closure call will occur. Model programs have been formalized and proven sound in the same paper [26]. In [16] it is pointed out that model programs do not work well when the control flow of a method is not statically known (an example of such a method is given). Even though model programs are attractive for certain use cases, we believe a greybox approach is contrary to Viper's blackbox approach.

2.1 Pre/Post Model

When it comes to approaches that are limited to first-order logic, the so called *Pre/-Post Model* is very prevalent. The *Pre/Post Model* introduces a placeholder *pre* and a placeholder *post* (sometimes other names are used). They represent a precondition and postcondition resp. of a closure that possibly points to an unknown function. These pre and post placeholder can be linked to concrete contracts. This is usually done when it is known to which method a closure points. The pre and post placeholder can also be used in assertions. This allows to restrict what contracts a closure can have. With suitable restrictions a closure call can then be verified even if it is unknown to which method the closure points.

The Pre/Post Model first appears in [10] using higher order logic. Much other work then builds on this idea using first-order logic. In the following, we briefly describe approaches that fall into this category.

In [25], Eiffel [22] is translated to Spec# [3]. They verify Eiffel's agents which are very similar to closures. The pre and post placeholders are called `$precondition` and `$postcondition`. For normal methods dynamic frames [15] are used. An abstract `$modifies` function is introduced to frame closures. Like the pre and post placeholders, the `$modifies` function is defined through local assumptions when agents are initialized.

The pre and post model is very well explained in [17] where a toy language is translated to Boogie [2]. A simplified version of dynamic frames [15] is used. The proposed framework supports many different use cases.

This work is then heavily used in [9]. There closure support is added to Dafny [19] which then gets translated to Boogie [2]. Dafny uses dynamic frames [15].

Chalice [20] is translated to Boogie in [1]. The pre and post placeholders are not directly used in Chalice. In essence a closure is directly restricted through contracts instead of indirectly through restrictions on the placeholders. In Boogie these placeholders are then introduced and used for verification of closures. To solve the framing problem Chalice uses fractional permissions [5].

In [16] a toy language is translated to Boogie. It can be seen as a follow up to [17] (same authors). In particular, their approach is formalized and proven sound.

Chalice is translated to Viper in [21]. Just like Nagini, Chalice uses fractional permissions. So the context of this work is very similar to this thesis. There are some limitations when it comes to old expressions and higher order functions.

2.2 Tools

In the following we focus on tools comparable to Viper (e.g., tools that use first-order logic). While many approaches can be encoded in languages supported by other tools, there are few tools that have dedicated support for closures.

JML [7] (The Java Modeling Language) implements model programs [26] to verify Java programs.

Dafny [19] has excellent support for pure closures. The way pure closures are supported in Dafny can be compared to the Pre/Post Model. Every pure function `f(...)` has an `f.requires(...)` function which would correspond to the pre placeholder. Since pure functions are opaque in Dafny, the post placeholder would correspond directly to the function `f(...)`. To support dynamic frames for pure closures every function also has an `f.reads(...)` function which returns the set of objects it reads.

VeriFast [14] is a verifier for C and Java programs. It uses separation logic to solve the framing problem. It supports C's function pointers which are C's version of closures. VeriFast has predicate families that are indexed by function pointers. This allows one to verify various use cases of closures. However, it is unclear how well closures are supported generally.

Chapter 3

Call Slots

3.1 Introduction

```
1  class Argument:
2
3      def __init__(self, parameter: int, result: int) -> None:
4          self.parameter = parameter  # type: int
5          self.result = result  # type: int
6
7          Ensures(Acc(self.parameter) and Acc(self.result))
8          Ensures(self.parameter == parameter and self.result == result)
9
10 def func(
11     g: Callable[[Argument, int, int], Tuple[int, int]],
12     a: Argument,
13     b: int,
14     c: int
15 ) -> Tuple[int, int, int]:
16
17     Requires(Acc(a.parameter))
18     Requires(Acc(a.result))
19     Ensures(Acc(a.parameter))
20     Ensures(Acc(a.result))
21
22     while a.parameter < b + c:
23         Invariant(Acc(a.parameter))
24         a.parameter = a.parameter * a.parameter
25
26     # g reads a.parameter and writes to a.result
27     d, e = g(a, b, c)
28
29     return 1 // a.result, sqrt(d), log(e)
```

FIGURE 3.1: Example code with a closure call.

Figure 3.1 shows Python code with a closure call (line 27). The code contains both type annotations and specifications for Nagini.

This code cannot be verified modularly because of the closure call on line 27. Viper cannot modularly know to which method g points. Therefor the precondition of g is unknown. However to verify this code, the author of the method `func` has to ensure that the precondition of g is established before calling it. Additionally, after calling g its return values are used in the return expressions on line 29. These expressions are only valid if $0 \neq$ `a.result` and $0 \leq$ `d` and $0 <$ `e`.

Before calling g there is a `while` loop. After the while loop, the negation of its condition is established: `a.parameter` $\geq$ `b` $+$ `c`.

From the comment on line 26, we can assume that the author of the method `func` wants to give `g` read permission to `a.parameter` and write permission to `a.result`. In Nagini these permissions can be expressed through the Python expression: `Acc(a.parameter, 1/2) and Acc(a.result)`.

Because we give back these permissions on line 19 and 20, it's only natural that we want the closure `g` to give back its access permissions to `a.parameter` and `a.result` too.

Based on the above observations, it seems reasonable that g's precondition should be established whenever the following assertion holds:

$$\texttt{Acc(a.parameter, 1/2) and Acc(a.result)} \\ \texttt{and a.parameter} \geq \texttt{b} + \texttt{c} \tag{3.1}$$

Furthermore, the following assertion should hold after the call to g:

$$\texttt{Acc(a.parameter, 1/2) and Acc(a.result)} \\ \texttt{and } 0 \neq \texttt{a.result and } 0 \leq \texttt{d and } 0 < \texttt{e} \tag{3.2}$$

With these two equations and the closure call on line 27 we can declare a *call slot*.

A *call slot* consists of a precondition (Equation 3.1), a call to a closure (line 27) and a postcondition (Equation 3.2). The precondition is an assertion that holds before the call and the postcondition is an assertion that holds after the call. Figure 3.2 shows how to declare a call slot for Nagini.

```
1  @CallSlot
2  def func_call_slot(
3      g: Callable[[Argument, int, int], Tuple[int, int]],
4      b: int, c: int
5  ) -> None:
6
7      @UniversallyQuantified
8      def uq(a: Argument) -> None:
9
10         Requires(Acc(a.parameter, 1 / 2) and Acc(a.result))
11         Requires(a.parameter >= b + c)
12
13         ret = g(a, b, c)
14
15         Ensures(Acc(a.parameter, 1 / 2) and Acc(a.result))
16         Ensures(a.result != 0 and ret[0] >= 0 and ret[1] > 0)
```

FIGURE 3.2: A call slot declaration for example 3.1.

A call slot is parametrized by normal variables (`g`, `b` and `c` on line 2 and 3 of Figure 3.2) and universally quantified variables (`a` on line 8 of Figure 3.2). Universally quantified variables can have any value of their respective types. A *call slot instance* is a call slot where the normal variables have been instantiated with values. A call slot instance is valid if all of the following conditions are met:

- The precondition of the call can be established from only the precondition of the call slot (for all values the universally quantified variables can have).

- The postcondition of the call slot can be established after the call which occured in a state as described by the precondition of the call slot (for all values the universally quantified variables can have).

Generally, it is unknown whether a call slot instance is valid or invalid. Later we will see that a call slot instance can be guaranteed to be valid through a proof. Call slots are always named. On line 2 of Figure 3.2 the name of the call slot is declared: `func_call_slot`.

Figure 3.3 shows how we can use the call slot `func_call_slot` to verify example 3.1.

```
1  def func(
2      g: Callable[[Argument, int, int], Tuple[int, int]],
3      a: Argument,
4      b: int,
5      c: int
6  ) -> Tuple[int, int, int]:
7
8      Requires(func_call_slot(g, b, c))  # new precondition
9      Requires(Acc(a.parameter))
10     Requires(Acc(a.result))
11     Ensures(Acc(a.parameter))
12     Ensures(Acc(a.result))
13
14     while a.parameter < b + c:
15         Invariant(Acc(a.parameter))
16         a.parameter = a.parameter * a.parameter
17
18     # g reads a.parameter and writes to a.result
19     d, e = ClosureCall(
20         g(a, b, c),
21         func_call_slot(g, b, c)(a)
22     )  # type: Tuple[int, int]
23
24     return 1 // a.result, sqrt(d), log(e)
```

FIGURE 3.3: Example 3.1 using the call slot `func_call_slot`.

First, we require that the call slot instance `func_call_slot(g, b, c)` has to be valid on line 8 of Figure 3.3. A call slot instance is expressed in Python through a call to the call slot where the normal variables instances are the arguments of that call. A call slot instance can be used as an assertion and as such will return whether the call slot instance is valid.

Because we cannot modularly know to which method the closure g points, we cannot modularly verify the closure call. We need to justify why it is ok to call the closure g on line 20. We do this with a `ClosureCall` annotation (line 19) which allows us to attach a justification to a closure call.

The `ClosureCall` annotation takes as first argument the closure call that needs to be justified and as second argument the justification for the closure call. There are two ways to justify a closure call. Firstly, through proving static dispatch which will be explained later in detail. Secondly, through a *call slot justification*.

The second case is used on line 21. There are two calls in the Python code. The first call (`func_call_slot(g, b, c)`) is a call slot instance (i.e., an instantiation of the call slots normal variables) and the second call (`(a)`) is the instantiation of the

universally quantified variables. A call slot instance where the universally quantified variables have been instantiated is a *call slot justification* and can be used as a justification for a closure call.

However, it is a valid justification if all of the following conditions are met:

- The call slot instance is valid.
- The call slot justification's precondition is established at that point.
- The call slot justification's call is equivalent to the closure call that needs to be justified.

Two closure calls are equivalent if all of the following conditions are met:

- the closure's values are equal.
- the first, second, ..., and last arguments of both calls are equal.

If two closure calls equivalent, they will have equivalent preconditions and postconditions.

Looking at the conditions of a valid justification, we observe that the second condition guarantees that the call slot justification's precondition is established. The first condition ensures that the closure call's precondition is established whenever the call slot justification's precondition is established. The third condition provides that the calls are equivalent and therefore have the same precondition and postcondition. This is sufficient to conclude that the precondition of the closure call will be established at that point. We have therefore justified the closure call through a call slot justification.

These conditions allow us to replace the unknown precondition of the closure with the known precondition of the call slot justification. The same reasoning can be used to see that we can replace the unknown postcondition of the closure with the known postcondition of the call slot justification.

In this case, all of these conditions are met and we can conclude that the closure call on line 20 is justified. Additionally, we can replace the unknown contracts of the closure with the known contracts of the call slot justification. This means we can exhale equation 3.1 and inhale equation 3.2. In particular, the inhaling allows us to verify that the expressions in the return statement are valid.

With the introduction of this call slot, we added the additional requirement to the method `func` that the call slot instance `func_call_slot(g, b, c)` has to be valid. This raises the question of how a caller of `func` can establish this additional precondition. Previously, we said this is done through a proof and we want to elaborate what this means now. Figure 3.4 shows code that calls the `func` method with a closure `concrete_g` that points to either the method `concrete_g_1` or the method `concrete_g_2`.

On line 50, we call the method `func` which requires full access permissions to both the `parameter` and `result` fields of the `Argument` object `a`. These preconditions are established by the constructor (`__init__`) of the `Argument` class (see Figure 3.1, line 7). However, `func` also requires that the call slot instance `func_call_slot(g, b, c)` is valid.

Generally, it is unknown whether a call slot instance is valid or invalid. However, with a call slot proof, we can guarantee that a call slot instance is valid. A *call slot proof* is a manual proof that proves a call slot instance valid.

The purpose of a call slot proof is to emulate a closure call of a call slot instance. The closure call should be emulated in a state as described by the precondition of the call slot instance and after the call the postcondition of the call slot instance should be established. If we can verify such a scenario, we have proven that the call slot instance is valid.

Lines 31 - 48 show an example of a call slot proof. A call slot proof always has to mention which call slot instance it tries to proof. This is done in the `CallSlotProof` decorator (line 31). The nested method declarations of a call slot proof have to match the nested method declarations of its corresponding call slot (lines 32 - 35). The contracts of the call slot (lines 37, 38, 47 and 48) can be repeated in the call slot proof to increase readability, but this is not necessary, because a verifier can look up the contracts of a call slot proof through its corresponding call slot.

The body of the call slot proof goes from line 40 to line 45. Its purpose is to emulate the closure call, i.e., it has to be a valid emulation of a closure call. A call slot proof's body is only a valid emulation if all of the following conditions are met:

- No program state is changed except through closure calls.
- When executing the body, exactly one closure call is executed, in all possible scenarios.
- All closure calls must be equivalent to the call slot's closure call.

The body of call slot proof from Figure 3.4 adheres to these restrictions.

Call slot proofs can use any assertions that are true where the proof occurs as long as these assertions do not require access permissions. On line 44, the assert statement will succeed, because the assertion that `concrete_g` is equal either to `concrete_g_1` or `concrete_g_2` is true at line 31 and it does not require any access permissions. The purpose of this is to allow a user to carry out a proof in a place where more information is available. The result of the proof (that a call slot instance is valid) can then be used in a different place where this information may no longer be available.

Access permissions are resources and they might not be available when a call slot instance is used to justify a closure call. That is why the proof body is executed in a state where all access permissions are no longer available except for access permissions provided by the precondition of the call slot instance.

The body of a call slot proof is executed in a program state that corresponds to the program state where the proof occurs. The body itself can change this program state. However, such a change of program state will be ignored after the call slot proof. After the call slot proof, the program will continue in the same state as before the call slot proof with the exception that it is guaranteed that the call slot instance is valid.

On line 41 and 45, two closure calls with a `ClosureCall` annotation occur. This time the justification (the second argument to the `ClosureCall` call) is not a call slot justification. In both cases the second argument is the name of a Python method. Earlier, we said that there are two kinds of justifications for closure calls. Either a call slot justification or static dispatch. These two closure calls use the static dispatch justification which means that we know to which methods the closures point on those lines. A static dispatch justification is valid if all of the following conditions are met:

- The closure points to the method specified in the second argument of the `ClosureCall` annotation.

```
1  def concrete_g_1(a: Argument, b: int, c: int) -> Tuple[int, int]:
2      Requires(Acc(a.parameter, 1 / 2) and Acc(a.result))
3      Requires(a.parameter >= b + c)
4      Ensures(Acc(a.parameter, 1 / 2) and Acc(a.result))
5      Ensures(a.result == -1 and Result()[0] == 0 and Result()[1] >= 1)
6
7      a.result = -1
8      return 0, a.parameter - b - c + 1
9
10
11 def concrete_g_2(a: Argument, b: int, c: int) -> Tuple[int, int]:
12     Requires(Acc(a.parameter, 1 / 2) and Acc(a.result))
13     Requires(a.parameter >= b + c)
14     Ensures(Acc(a.parameter, 1 / 2) and Acc(a.result))
15     Ensures(a.result == 1 and Result()[0] >= 0 and Result()[1] == 1)
16
17     a.result = 1
18     return a.parameter - b - c, 1
19
20
21 def client() -> None:
22
23     if choice():  # non-deterministic choice
24         concrete_g = concrete_g_1
25     else:
26         concrete_g = concrete_g_2
27
28     a = Argument(2, 2)
29     b, c = 2, 3
30
31     @CallSlotProof(func_call_slot(concrete_g, b, c))
32     def func_call_slot_proof(concrete_g: G_Type, b: int, c: int) -> None:
33
34         @UniversallyQuantified
35         def uq(a: Argument) -> None:
36
37             # Requires(Acc(a.parameter, 1 / 2) and Acc(a.result))
38             # Requires(a.parameter >= b + c)
39
40             if concrete_g == concrete_g_1:
41                 ret = ClosureCall(concrete_g(a, b, c), concrete_g_1
42                     )  # type: Tuple[int, int]
43             else:
44                 assert concrete_g == concrete_g_2
45                 ret = ClosureCall(concrete_g(a, b, c), concrete_g_2)
46
47             # Ensures(Acc(a.parameter, 1 / 2) and Acc(a.result))
48             # Ensures(a.result != 0 and ret[0] >= 0 and ret[1] > 0)
49
50     func(concrete_g, a, b, c)
```

FIGURE 3.4: A possible client of the `func` method of example 3.1.

- The precondition of that method is established at that point.

If we can prove static dispatch for a closure, the closure actually does not point to an unknown method, but instead to a known method. Therefore, we can use the contracts of that method to verify the closure call.

As we know, call slots can have both normal variables and universally quantified variables. In this example we used a universally quantified variable for the `Argument` object which is initialized on line 28 of Figure 3.4. Then the call slot proof occurs on line 31 of the same Figure. On line 50, it is passed to the `func` method which modifies the `Argument` object on line 16 of Figure 3.3. On line 21 of Figure 3.3 the universally quantified variable `a` is instantiated with this `Argument` object. Because variable `a` is universally quantified in the call slot, the call slot instance is valid for all possible `Argument` instances. So even though the `Argument` object was modified between the call slot proof and the closure call, the result of the call slot proof (the call slot instance) is still valid where the closure call occurs.

The call slot uses normal variables for the variables `g`, `b` and `c`. The call slot proof was done for the call slot instance `func_call_slot(concrete_g, b, c)`. That is, the instance is only valid for the values of `concrete_g`, `b` and `c`. If any of those variables' values changed, the call slot instance would no longer be valid for those variables. On the other hand, the call slot proof could make use of information about those values. In particular, that `concrete_g` points to either of `concrete_g_1` or `concrete_g_2`. The proof could have also used the information that `b` $==$ 2 and `c` $==$ 3, but it was not necessary for this proof.

Universally quantified variables allow us to use call slots for values that can change between a call slot proof and where the resulting call slot instance is used in a `ClosureCall`. This is also true for values that only exist where the `ClosureCall` occurs (but not where the call slot proof is carried out). Normal variables allow us to use information about their values in a call slot proof that is only available where the proof occurs.

3.2 Pure Call Slots

In Nagini one can declare a method to be pure with the `Pure` decorator.

```
1  class Argument:
2
3      def __init__(self, value_a: int, value_b: int) -> None:
4          self.value_a = value_a
5          self.value_b = value_b
6          Ensures(Acc(self.value_a) and Acc(self.value_b))
7          Ensures(self.value_a == value_a and self.value_b == value_b)
8
9  @Pure
10 def add(arg: Argument, x: int) -> int:
11     Requires(Acc(arg.value_a))
12     Ensures(Result() == arg.value_a + x)
13     return arg.value_a + x
```

FIGURE 3.5: An example of a pure method (lines 9 - 13).

A method that is pure has at least all of the following properties:

- It is side effect free: No program state is changed when a pure method is called.

- It is deterministic: It will always return the same if it is called with the same values.
- It can be used in contracts and assertions.

Call slots do not have these properties, because they support impure method (methods that are not pure). Call slot proofs can be carried out with pure methods, but when the resulting call slot instance is used as a call slot justification, the closure call will not have the above properties.

If a `ClosureCall` uses static dispatch as a justification, Nagini can look up the method and find out whether it is a pure or impure method. If it is a pure method, then the closure call will also have the above properties.

For a closure call that is guaranteed to be pure and that we want to justify with a call slot justification, we can use pure call slots. So far we have introduced normal call slots (for impure methods). Pure call slots are very similar to normal call slots with a few key differences. With these differences a `ClosureCall` will also have the three properties of pure methods described above when a pure call slot justification is used. In the following we will describe these differences.

```
1  f_type = Callable[[Argument, int], int]
2
3  @Pure
4  @CallSlot
5  def add_or_mul(f: f_type) -> None:
6
7      @UniversallyQuantified
8      def uq(arg: Argument, x: int) -> None:
9          Requires(Acc(arg.value_a) and Acc(arg.value_b))
10
11         y = f(arg, x)
12
13         Ensures(y == arg.value_a + x or y == arg.value_b * x)
```

FIGURE 3.6: An example of a pure call slot.

A pure call slot can be declared like a normal call slot, but it has to have the `Pure` decorator (line 3, Figure 3.6).

A pure call slot proof's body has to be a valid emulation of a pure closure call. It is only a valid emulation of a pure closure call if all of the following conditions are met:

- No program state is changed.
- When executing the body, at least one pure closure call is executed that is equivalent to the call slot's closure call, in all possible scenarios.

A closure call is pure if it is either justified by a pure call slot justification or by proving static dispatch to a pure method.

Figure 3.7 shows how the pure `add` method and the pure `add_or_mul` call slot can be used to verify a pure closure call. The assertion on line 35 verifies that the closure call is indeed deterministic. Even though the closure has `write` access to the `value_a` and `value_b` fields, we can verify that they are unchanged on lines 36 and 37. Thus we could verify the closure call to be side effect free. On line 38, we are able to call a pure closure inside a contract.

```
1  @Pure
2  def mul(arg: Argument, x: int) -> int:
3      Requires(Acc(arg.value_b))
4      Ensures(Result() == arg.value_b * x)
5      return arg.value_b * x
6
7  def client() -> None:
8
9      arg = Argument(1, 2)
10
11     if choice():  # non-deterministic choice
12         f = add
13     else:
14         f = mul
15
16     @CallSlotProof(add_or_mul(f))
17     def add_or_mul_proof(f: f_type) -> None:
18
19         @UniversallyQuantified
20         def uq(arg: Argument, x: int) -> None:
21             # Requires(Acc(arg.value_a) and Acc(arg.value_b))
22
23             if f == add:
24                 y = ClosureCall(f(arg, x), add)  # type: int
25             else:
26                 y = ClosureCall(f(arg, x), mul)
27
28             # Ensures(y == arg.value_a + x or y == arg.value_b * x)
29
30     assert arg.value_a == 1
31     assert arg.value_b == 2
32     y1 = ClosureCall(f(arg, 3), add_or_mul(f)(arg, 3))  # type: int
33     y2 = ClosureCall(f(arg, 3), add_or_mul(f)(arg, 3))  # type: int
34     assert y1 == 4 or y2 == 6
35     assert y1 == y2
36     assert arg.value_a == 1
37     assert arg.value_b == 2
38     assert y1 == ClosureCall(f(arg, 3), add_or_mul(f)(arg, 3))
```

FIGURE 3.7: An example of a pure call slot proof and pure closure calls.

Previously, we said that a call slot proof (normal, not pure) has to have a body that is a valid emulation of a closure call. The definition of a valid emulation of a closure call did not take pure closure calls into consideration. The following definition takes pure closure calls into consideration. A call slot proof's body (normal, not pure) is a valid emulation of a closure call if all of the following conditions are met:

- No program state is changed except through closure calls.
- When executing the body, exactly one impure and no pure closure calls are executed or at least one pure and no impure closure call are executed, in all possible scenarios. For both cases those closure calls have to be equivalent to the call slot's closure call.
- All impure closure calls must be equivalent to the call slot's closure call.

3.3 Old Expressions

Nagini supports old expressions in postconditions of impure methods. The `inc` method makes use of old expressions to specify its behavior in the following example (lines 18 and 19):

```
1  class Argument:
2
3      def __init__(self, parameter: int, result: int) -> None:
4          self.parameter = parameter  # type: int
5          self.result = result  # type: int
6
7          Ensures(Acc(self.parameter) and Acc(self.result))
8          Ensures(self.parameter == parameter and self.result == result)
9
10
11 inc_type = Callable[[Argument], Optional[object]]
12
13
14 def inc(arg: Argument) -> Optional[object]:
15     Requires(Acc(arg.parameter) and Acc(arg.result))
16
17     Ensures(Acc(arg.parameter) and Acc(arg.result))
18     Ensures(arg.result == Old(arg.result) + arg.parameter)
19     Ensures(arg.parameter == Old(arg.parameter))
20
21     arg.result = arg.result + arg.parameter
22
23     return None
```

FIGURE 3.8: An example of an impure method using old expressions.

Since a closure could point to the `inc` method, closures should have support for old expressions too. Call slots support old expressions naturally. In the following example a call slot is declared with old expressions in its postcondition (lines 8 and 9).

```
1 @CallSlot
2 def inc_call_slot(f: inc_type, arg: Argument) -> None:
3     Requires(Acc(arg.parameter) and Acc(arg.result))
4
5     f(arg)
6
7     Ensures(Acc(arg.parameter) and Acc(arg.result))
8     Ensures(arg.result >= Old(arg.result) + arg.parameter)
9     Ensures(arg.parameter == Old(arg.parameter))
```

FIGURE 3.9: A call slot with old expressions.

Figure 3.10 shows that call slot proofs and `ClosureCalls` also support old expressions. The assertion on line 22 verifies, because we can use old expressions with closures.

```
1  def test() -> None:
2
3      arg = Argument(1, 2)
4
5      arg.result = 20
6      arg.parameter = 50
7
8      f = inc
9
10     @CallSlotProof(inc_call_slot(inc, arg))
11     def inc_proof(f: inc_type, arg: Argument) -> None:
12         # Requires(Acc(arg.parameter) and Acc(arg.result))
13
14         ClosureCall(f(arg), inc)
15
16         # Ensures(Acc(arg.parameter) and Acc(arg.result))
17         # Ensures(arg.result >= Old(arg.result) + arg.parameter)
18         # Ensures(arg.parameter == Old(arg.parameter))
19
20     ClosureCall(f(arg), inc_call_slot(f, arg)())
21
22     assert arg.result >= 70 and arg.parameter == 50
```

FIGURE 3.10: Code that uses a call slot with old expressions.

3.4 Viper Encoding

In the following the encoding into Viper will be explained. We will first explain how to encode methods as first-class citizen so that they can be, e.g., assigned to variables. We will then describe how call slots are encoded. We will continue with the encoding of the `ClosureCall` annotation and finally show how call slot proofs are encoded.

During translation many names will be introduced for variables, functions, methods, etc in Viper. These names could collide with existing names. Nagini provides means to avoid such collisions which are used by our implementation. For all names introduced one can assume that collisions with existing names are guaranteed to be avoided.

3.4.1 Methods as First-Class Citizen

Figure 3.11 shows code where methods are assigned to variables. The Python methods `concrete_g_1` and `concrete_g_2` will be translated to Viper methods. However, in Viper methods are not first-class citizen which means we cannot assign them to variables.

To work around that we introduce a Viper domain `Function`. For each method we add a function to that domain which returns a unique value of that domain. Those domain functions are on lines 2-4 in Figure 3.12. Then whenever a method's value is used, we use the corresponding domain function. On lines 13 and 15, we see the values used in Viper to encode the values of `concrete_g_1` and `concrete_g_2`.

Furthermore, the variable representing a closure will be of type `Function` in Viper. Line 8 shows the corresponding variable in Viper for the closure `concrete_g`.

```python
def concrete_g_1(a: Argument, b: int, c: int) -> Tuple[int, int]:
    # ...

def concrete_g_2(a: Argument, b: int, c: int) -> Tuple[int, int]:
    # ...

def client() -> None:

    if choice():  # non-deterministic choice
        concrete_g = concrete_g_1
    else:
        concrete_g = concrete_g_2

    # ...
```

FIGURE 3.11: Code from Figure 3.4 where methods are assigned to variables.

```
domain Function {
  unique function get_concrete_g_1(): Function
  unique function get_concrete_g_2(): Function
  unique function get_client(): Function
}

method client() {
  var concrete_g: Function

  var choice_result: Bool
  // choice_result = choice()
  if (choice_result) {
    concrete_g := get_concrete_g_1()
  } else {
    concrete_g := get_concrete_g_2()
  }
}
```

FIGURE 3.12: Encoding of method values in Viper based on the code from Figure 3.11.

3.4.2 Call Slots

For each call slot we introduce an encoding for call slot instances and an encoding for call slot justifications. We will first focus on call slot instances and then on call slot justifications. Figure 3.13 shows the running example of this section.

A call slot instance is encoded as an abstract boolean function in Viper. Its parameters are all normal variables of a call slot. Figure 3.14 shows the Viper code for the call slot instance of our running example. The return value represents whether a call slot instance is valid (true) or invalid (false). Requiring a call slot instance to be valid gets translated to requiring the abstract boolean function to be true. Checking whether a call slot instance is valid gets translated to checking whether the abstract boolean function is true. The boolean function is abstract because by default it is unknown whether a call slot instance is valid or not. In section 3.4.4 we will explain how a call slot proof adds the knowledge that a call slot instance is valid.

A call slot justification is a call slot instance where the universally quantified variables are instantiated. It can be used to justify a closure call in a `ClosureCall`

```
1  @CallSlot
2  def func_call_slot(
3      g: Callable[[Argument, int, int], Tuple[int, int]],
4      b: int, c: int
5  ) -> None:
6
7      @UniversallyQuantified
8      def uq(a: Argument) -> None:
9
10         Requires(Acc(a.parameter, 1 / 2) and Acc(a.result))
11         Requires(a.parameter >= b + c)
12
13         ret = g(a, b, c)
14
15         Ensures(Acc(a.parameter, 1 / 2) and Acc(a.result))
16         Ensures(a.result != 0 and ret[0] >= 0 and ret[1] > 0)
```

FIGURE 3.13: The call slot from Figure 3.2.

```
1  function func_call_slot(g: Function, b: Int, c: Int): Bool
```

FIGURE 3.14: Encoding of the call slot instance `func_call_slot`.

annotation. For each call slot a helper method is added for the translation of the `ClosureCall` annotation. Figure 3.15 shows the helper method for our call slot.

The helper method has the following parameters: All normal variables of the call slot, all universally quantified variables of the call slot, the closure and all the parameters of the call (line 13 of Figure 3.13). The normal variables appear on line 3 of Figure 3.15. On line 5 we declared the universally quantified variables and on line 7 are the parameters of the call and the closure.

We want to elaborate on the last set of parameters (line 7). This helper method will be used when a `ClosureCall` annotation is justified with a call slot justification from the call slot `func_call_slot`. The `ClosureCall` annotation takes two parameters: The closure call and the justification. It is only valid if the closure call is equivalent to the call of the call slot justification. The closures' value and its arguments will be the values used for this last set of parameters (line 7). On lines 12 - 15 we can see that we check whether the closures' value and the arguments of the closure call are the same as the values of `func_call_slot`'s call (line 13 of Figure 3.13).

On line 10 of Figure 3.15 we require that the call slot instance is valid. On the lines 17 and 18 we require the precondition of the call slot and on the lines 20-23 we ensure the postcondition of the call slot (the function `tuple___getitem__` is used to get the values of tuples).

We observe that this helper method's precondition corresponds to all conditions required for a call slot justification to be valid. Furthermore, its postcondition corresponds to the call slot's postcondition.

This helper method naturally supports old expressions in call slots' postconditions. If the call slot is pure, then the translation works the same except that a Viper function is used instead of a Viper method. Using a Viper function we naturally get the properties of a pure function as described earlier.

```
1  method func_call_slot_apply(
2    /* normal variables */
3    g: Function, b: Int, c: Int,
4    /* universally quantified variables */
5    a: Ref,
6    /* closure value and the arguments to the closure call */
7    call_target: Function, call_arg_1: Ref, call_arg_2: Int, call_arg_3: Int
8  ) returns (ret: Ref)
9    /* call slot instance has to be valid */
10   requires func_call_slot(g, b, c)
11   /* closure call matches call slot's call */
12   requires call_target == g
13   requires call_arg_1 == a
14   requires call_arg_2 == b
15   requires call_arg_3 == c
16   /* precondition of call slot */
17   requires acc(a.parameter, 1/2) && acc(a._result)
18   requires a.parameter >= b + c
19   /* postcondition of call slot */
20   ensures acc(a.parameter, 1/2) && acc(a._result)
21   ensures a._result != 0
22   ensures tuple___getitem__(ret, 0) >= 0
23   ensures tuple___getitem__(ret, 1) > 0
```

FIGURE 3.15: The helper method for `func_call_slot`'s call slot justifications.

3.4.3 Closure Calls

Closure calls always have to be justified except in call slot declarations. With the `ClosureCall` annotation we can justify a closure call. There are two kinds of justifications: Call slot justifications or proving static dispatch. We will first explain the former and then the later.

```
1      # ...
2
3      # g reads a.parameter and writes to a.result
4      d, e = ClosureCall(
5          g(a, b, c),
6          func_call_slot(g, b, c)(a)
7      )  # type: Tuple[int, int]
8
9      # ...
```

FIGURE 3.16: A closure call justified with a call slot justification from Figure 3.3.

In section 3.4.2 we explained that for each call slot a helper method is introduced. For our running example the method `func_call_slot_apply` was introduced. Whenever a closure call is justified by a call slot justification, the helper method of that call slot will be called. Figure 3.16 shows a code snippet where a closure call is justified by a call slot justification. We observe that all values needed for the helper method are available where the `ClosureCall` annotation occurs.

Figure 3.17 shows the corresponding viper encoding. The arguments on line 4 and 6 are extracted from Figure 3.16 line 6. The arguments on line 8 are extracted from Figure 3.16 line 5.

With this encoding all conditions for this kind of justification are correctly checked. Nagini will then generate code to extract the values of the tuple `ret` into the variables `d` and `e`.

```
1  /* ... */
2  ret := func_call_slot_apply(
3    /* call slot justification's normal variables */
4    g, b, c,
5    /* call slot justification's universally quantified variables */
6    a,
7    /* values of closure call */
8    g, a, b, c
9  )
10 /* ... */
```

FIGURE 3.17: Viper encoding of the code snippet from Figure 3.16.

The other kind of justification we can use in a `ClosureCall` annotation is proving static dispatch. In this case we just have to check that the closure indeed points to the method specified in the `ClosureCall` annotation and then we can directly call that method. However, we will also use a helper method for this simple case. That is, we will generate an abstract method with the same arguments, contracts and return values as the method to which the closure points. But additionally we will add a parameter for the closure's value and a precondition that the closure has to point to that method. The `ClosureCall` annotation will then be translated to calling that method with the corresponding arguments of the closure call and the closure's value.

If we justify a closure call by proving static dispatch to a pure method, we generate the same helper method, with the difference that we generate it as a Viper function instead.

With this encoding we translate every `ClosureCall` annotation to either a method call or function call in Viper (depending on whether it is an impure or pure closure call). The checks that the justification is valid are done in the preconditions. This allows us to call pure closures in contracts, because they will be translated to function calls in Viper which are allowed to occur in contracts.

3.4.4 Call Slot Proofs

To explain call slot proofs, we will first explain how to create the correct environment for a proof and then how to translate a proof body inside such an environment. At the end of section 3.2 we specified the conditions for valid call slot proof bodies of pure and impure call slots. The translation ensures that we verify that those conditions are fulfilled.

We will use Figure 3.18 as our running example to explain how we create an environment for a call slot proof. That code will be translated to the Viper code of Figure 3.19.

We first declare variables for the proof's normal variables (line 3 - 5, Figure 3.19). Those variables will be initialized with the values of the call slot instance we want to prove valid (lines 7 - 9). Then we create a while loop with a non-deterministic choice as condition (lines 11 and 12). Because the verifier cannot know whether the condition will be true or false, the verifier is forced to verify both the loop body and the code after the loop. Because the condition never changes, the code after the loop

will be verified as if the loop never executed. Since there are no loop invariants all access permissions are lost inside the loop body.

```
1     # ...
2     @CallSlotProof(func_call_slot(concrete_g, b, c))
3     def func_call_slot_proof(concrete_g: G_Type, b: int, c: int) -> None:
4
5         @UniversallyQuantified
6         def uq(a: Argument) -> None:
7
8             # Requires(Acc(a.parameter, 1 / 2) and Acc(a.result))
9             # Requires(a.parameter >= b + c)
10
11            # proof body ...
12
13            # Ensures(Acc(a.parameter, 1 / 2) and Acc(a.result))
14            # Ensures(a.result != 0 and ret[0] >= 0 and ret[1] > 0)
15    # ...
```

FIGURE 3.18: The call slot proof from Figure 3.4.

Inside the loop we declare the universally quantified variables (line 14) and a return variable for the closure call if necessary. Return variables of closure calls inside call slot proofs have certain restrictions which will be explained later.

We continue with inhaling the precondition (lines 18 and 19) and then declare a label that we can later use if old expressions occur in the postcondition of the call slot (line 21). Furthermore, we declare and initialize a counter variable for each pure and impure closure calls (lines 23 - 26). At this point we insert the translated proof body (line 28) which is restricted (those restrictions are explained later).

After the proof body, we exhale the postcondition of the call slot (lines 31 - 34). If it contains old expressions, they will be translated to labelled old expressions using the label declared earlier. The loop body then finishes with checking the closure call counters (lines 36 - 42). If the proof is for a pure call slot, only the expression on lines 40 and 41 is used instead, so we guarantee that no impure closure calls occur. We observe that impure call slots can be proven with pure closures.

Finally, we add the knowledge that the call slot instance is valid after the proof (line 45).

We now describe how a proof body gets translated. We translate the proof body normally and put it inside a proof environment as described above. During translation we check that the body is a valid proof body. One of the conditions for a proof body to be valid is that no program state gets changed except through impure closure calls (see section 3.2). This condition is generally hard to check. We enforce a stronger version of this condition with more restrictive checks. However, we note that it is possible to be less restrictive and allow more proofs that would still be valid. Nevertheless, our version allows for strong proofs and we believe it supports the majority of use cases.

A proof body is only valid if all of its statements are one of the following:

- `Fold` or `Unfold` statements.
- `Assert` or `Assume` statements.
- `if`, `elif` and `else` clauses where every condition is side-effect free and each body is again a valid proof body.

- A call slot proof.
- A closure call with or without a single target variable for the return value. If a variable is used for the return value, it must always be the same.

```
/* ... */
/* introduce proof variables */
var proof_concrete_g: Function
var proof_c: Int
var proof_b: Int
/* extract proof variables */
proof_concrete_g := concrete_g
proof_b := b
proof_c := c
/* uninitialized boolean */
var proof_non_deterministic_choice: Bool
while (proof_non_deterministic_choice) {
  /* universally quantified variables */
  var proof_a: Ref
  /* return variable if necessary */
  var proof_ret: Ref
  /* call slot's precondition */
  inhale acc(proof_a.parameter, 1/2) && acc(proof_a._result)
  inhale proof_a.parameter >= proof_b + proof_c
  /* old label */
  label proof_old
  /* proof call counters */
  var proof_impure_call_counter: Int
  var proof_pure_call_counter: Int
  proof_impure_call_counter := 0
  proof_pure_call_counter := 0

  /* proof body ... */

  /* call slot's postcondition */
  exhale proof_a._result != 0
  exhale tuple___getitem__(proof_ret, 0) >= 0
  exhale tuple___getitem__(proof_ret, 1) > 0
  exhale acc(proof_a.parameter, 1/2) && acc(proof_a._result)
  /* check call counters */
  assert ((
      proof_impure_call_counter == 1 &&
      proof_pure_call_counter == 0
    ) || (
      proof_impure_call_counter == 0 &&
      proof_pure_call_counter >= 1
  ))
}
/* Add knowledge that call slot instance is valid */
assume func_call_slot(concrete_g, b, c)
/* ... */
```

FIGURE 3.19: Viper encoding of the code snippet from Figure 3.18.

Lastly, we add Viper code that updates the closure call counters wherever a closure call occurs as a statement. If a pure closure call occurs, we increment the pure closure call counter (`proof_pure_call_counter`) if the closure call matches the closure call of the call slot. If an impure closure call occurs we assert that the closure call matches the call slot's closure call and increment the impure closure call counter (`proof_impure_call_counter`).

Checking whether a closure call of a proof matches the closure call of the call slot can be done by substituting the call slot's variables (both normal variables and universally quantified variables) with the proof's variables for the closure call's value and arguments of the call slot. Then we can check whether these expression are equal to the closure call's value and arguments of the proof.

Chapter 4

Parametric Assertions

4.1 Motivation

To verify closures with call slots it is necessary to declare preconditions and postconditions. That is, preconditions and postconditions have to be declared with known assertions. However, there are scenarios where this is not desirable.

```
1  class State:
2
3      def __init__(self, value: int, ) -> None:
4          self.value = value
5          Ensures(Acc(self.value) and self.value == value)
6
7
8  f_type = Callable[[State], Optional[object]]
9
10
11 def n_times(f: f_type, n: int, s: State) -> None:
12     Requires(0 <= n)
13
14     i = 0
15     while i < n:
16         Invariant(0 <= i and i <= n)
17
18         f(s)
19         i += 1
```

FIGURE 4.1: A method that executes a closure n times.

In the above code, a closure `f` is called `n` times. We would like to verify the `n_times` method using call slots. The closure call on line 18 should be justified with a call slot justification. So first, we have to declare a call slot. The question now arises what will the precondition and postcondition be for this call slot. Preferably, the contracts of the call slots should support as many closures as possible.

When verifying the `n_times` method there is another problem. To establish a precondition of a call slot justification on line 18, the loop invariant on line 16 has to be adjusted. With the current loop invariant we can only establish `0 <= i and i < n` on line 18. To support more useful preconditions, we need a better loop invariant that accomodates for the call slot's contracts.

4.2 Parametric Assertions for Methods and Call Slots

Both of these problems can be solved with *parametric assertions*. A *parametric assertion* is an assertion that is a parameter of a method or call slot. With parametric assertions methods and call slots can be parametrized with assertions. Inside the method or call slot the assertion represented by the parametric assertion is unknown (similar to abstract predicates). A client of the method or call slot has to specify what assertion is used for the parametric assertion.

```
1  inv_type = Assertion[State, int]
2
3
4  @CallSlot
5  def n_times_slot(f: f_type, n: int, inv: inv_type) -> None:
6
7      @UniversallyQuantified
8      def uq(s: State, i: int) -> None:
9          Requires(0 <= i and i < n)
10         Requires(inv(s, i))
11
12         f(s)
13
14         Ensures(inv(s, i + 1))
15
16
17 def n_times(f: f_type, n: int, s: State, inv: inv_type) -> None:
18     Requires(0 <= n)
19     Requires(inv(s, 0))
20     Requires(n_times_slot(f, n, inv))
21     Ensures(inv(s, n))
22
23     i = 0
24     while i < n:
25         Invariant(0 <= i and i <= n)
26         Invariant(inv(s, i))
27
28         ClosureCall(f(s), n_times_slot(f, n, inv)(s, i))
29         i += 1
```

FIGURE 4.2: Verifying the `n_times` method using parametric assertions.

We will now explain how parametric assertions can be used to solve the problems highlighted in the previous section (Section 4.1).

On line 17 of Figure 4.2, we parametrize the method `n_times` with an assertion. That is, we introduce the parametric assertion `inv` to this method. On line 26, we use the assertion `inv(s, i)` as an additional loop invariant. The call slot `n_times_slot` is also parametrized by an assertion (line 5). The call slots' precondition and postcondition use this parametric assertion (lines 10 and 14). Finally, the `n_times` method specifies new contracts on lines 19 - 21. We additionally requires `inv(s, 0)` as well as a call slot instance. In the postcondition we ensure `inv(s, n)`.

With these changes we introduced the parametric assertion `inv` and added it as a loop invariant (`inv(s, i)`). We added `inv(s, 0)` as a precondition which will establish the invariant when entering the loop. We also require the call slot instance `n_times_sot(f, n, inv)`. With this call slot instance the closure is required to establish the invariant of the next iteration (`inv(s, i + 1)`) given the invariant of the

previous iteration (`inv(s, i)` and $0 \leq$ `i` and `i` $\leq$ `n`). Finally, after the closure has been called `n` times, the loop invariant `inv(s, i)` will establish the postcondition of `n_times` because we can show that `i` $=$ `n`.

4.3 Parametric Assertions for Method Clients and Call Slot Proofs

```
1  @Predicate
2  def count_inv(s: State, i: int) -> bool:
3      return Acc(s.value) and s.value == i
4
5
6  def count(s: State) -> Optional[object]:
7      Requires(Acc(s.value))
8      Ensures(Acc(s.value) and s.value == Old(s.value) + 1)
9
10     s.value += 1
11
12     return None
13
14
15 def n_times_client() -> None:
16
17     s = State(0)
18
19     f = count
20
21     @CallSlotProof(n_times_slot(f, 23, count_inv))
22     def n_times_slot(f: f_type, n: int, inv: inv_type) -> None:
23
24         @UniversallyQuantified
25         def uq(s: State, i: int) -> None:
26             # Requires(0 <= i and i < n)
27             # Requires(inv(s, i))
28             Unfold(inv(s, i))
29
30             ClosureCall(f(s), count)
31
32             Fold(inv(s, i + 1))
33             # Ensures(inv(s, i + 1))
34
35     assert s.value == 0
36     Fold(count_inv(s, 0))
37     n_times(f, 23, s, count_inv)
38     Unfold(count_inv(s, 23))
39     assert s.value == 23
```

FIGURE 4.3: A possible client of `n_times`.

When calling a method with a parametric assertion, the parametric assertion is instantiated with an assertion at the call site. At the call site we can take advantage of the knowledge what assertion is used for a parametric assertion. This also applies to call slot proofs. When proving a call slot with a parametric assertion we know what assertion we use for its parametric assertion. In the proof we can use this knowledge. Figure 4.3 demonstrates this.

On line 37 we call `n_times` with the closure `f`. On line 19 the `count` method is assigned to this closure. The `count` method simply increments the `value` field of a `State` object (line 10). The method `n_times` is called with a `State` object whose `value` field is initially 0. Since this field is incremented in each iteration, a possible loop invariant for the `n_times` method would be `s.value` $=$ `i`. We use the `count_inv` predicate for the parametric assertion `inv` when calling the `n_times` method. This predicate uses the proposed loop invariant together with the necessary access permissions.

With these parameters, the `n_times` method requires the call slot instance `n_times_client(f, 23, count_inv)`. This instance is proven on the lines 21 - 33. In this proof, we unfold the parametric assertion `inv` on line 28. At first it might seem wrong that we can unfold a parametric assertion, since parametric assertions are unknown inside methods and call slots. However, in the proof we can use the knowledge that `inv` is instantiated with the predicate `count_inv`. That is, inside the proof the parametric assertion is not unknown. Instead we know its instantiation. On line 30, the closure call is justified by proving static dispatch to the `count` method. Its precondition is established by the `count_inv(s, i)` predicate which we unfolded on line 28. We can then fold the `count_inv(s, i + 1)` predicate on line 32. This succeeds because of the `count` method's postcondition.

On line 36 we fold the `count_inv(s, 0)` predicate which will establish the `inv(s, 0)` precondition of `n_times`. This works because at the call site we know that `inv` is instantiated with `count_inv`. Because $0 \leq 23$ all preconditions are established for the `n_times` method call. The postcondition then gives us `inv(s, 23)` or `count_inv(s, 23)` when taking the instantiation of the parametric assertion into consideration. After we unfold this predicate we can indeed verify that `s.value` $= 23$ on line 39.

4.4 Viper Encoding

In the following we will explain how we can encode parametric assertions in Viper. For our encoding we impose additional restrictions which allow a straight forward encoding which we describe first. Then we show how we can encode assertions as first-class citizen in Viper. We continue to explain parametric assertions for methods and call slots. Finally, we will explain how parametric assertions are encoded for clients of method and call slot proofs.

Parametric assertions work for pure and impure methods and also for pure and impure call slots. In the following we will refer to both pure and impure methods simply as methods and we will refer to both pure and impure call slots simply as call slots.

In our encoding we will introduce many new names. Again, we assume that possible collisions with existing names are handled separately. Nagini provides means for this.

4.4.1 Simple Parametric Assertions

For our version of parametric assertions we impose additional restrictions that allow a straight forward encoding. With these restrictions it may help to think of parametric assertions as a simple version of Java's type generics. Simple parametric assertions can be seen as assertion generics. We list here the restrictions for simple parametric assertions.

Variables of assertion type:

- can only be parameters of methods or call slots.
- cannot be assigned to.
- can only be instantiated with predicates or variables of assertion type.
- can only be used to instantiate other parametric assertions. However, parametric assertions can be evaluated in contracts.

4.4.2 Assertions as First-Class Citizen

In Viper assertions are not first-class citizen which means we cannot assign assertions to variables. To work around this we introduce a domain `Assertion`. For every parameter of a method or call slot that is an assertion, we introduce a unique function that represents the value of that parameter. Since parametric assertions can be instantiated with predicates, we also introduce such a function for each predicate.

Figure 4.4 shows this domain for the running example of this chapter. On line 2, we introduce a function for the assertion from Figure 4.2, line 17. The function on line 3, represents the assertion from Figure 4.2, line 5. The last function on line 4, was generated from the predicate of Figure 4.3, line 2.

```
1 domain Assertion {
2   unique function get_n_times_inv(): Assertion
3   unique function get_n_times_slot_inv(): Assertion
4   unique function get_count_inv(): Assertion
5 }
```

FIGURE 4.4: The `Assertion` domain representing values of assertions.

4.4.3 Methods

Methods only have to be verified if they have a body. So we will only focus on how to translate methods with parametric assertions and a body. Translation of methods without a parametric assertions is unaffected by parametric assertions.

```
1  inv_type = Assertion[State, int]
2  def n_times(f: f_type, n: int, s: State, inv: inv_type) -> None:
3      Requires(0 <= n)
4      Requires(inv(s, 0))
5      Requires(n_times_slot(f, n, inv))
6      Ensures(inv(s, n))
7
8      i = 0
9      while i < n:
10         Invariant(0 <= i and i <= n)
11         Invariant(inv(s, i))
12
13         ClosureCall(f(s), n_times_slot(f, n, inv)(s, i))
14         i += 1
```

FIGURE 4.5: A method with a body and a parametric assertion.

Figure 4.5 shows a method with a parametric assertion. Its corresponding viper encoding is shown in Figure 4.6. In the previous section we explained that we introduce a function to the `Assertion` domain for each parametric assertion of a method or call slot. So for the `inv` variable we will also have a function representing its value. To encode this, we add the additional precondition that `inv`'s value is equal to the value of its `Assertion` domain function (line 6 of Figure 4.6).

The parametric assertion `inv` can be instantiated either with a predicate or another parametric assertion. Inside the method `n_times` the assertion that `inv` represents is unknown. This is best encoded with an abstract predicate. For every parametric assertion of a method we introduce an abstract predicate that represents its value when it is evaluated (e.g., on line 4 or 6). Line 1 of Figure 4.6 shows the generated abstract predicate for the `inv` assertion.

Finally, whenever `inv` is evaluated we simply use the generated abstract predicate instead. This happens on the lines 4, 6 and 11 of Figure 4.5 which get translated to the lines 4, 7 and 14 resp. of Figure 4.6.

```
1  predicate n_times_inv(r: Ref, x: Int)
2  method n_times(f: Function, n: Int, s: Ref, inv: Assertion)
3    requires 0 <= n
4    requires n_times_inv(s, 0)
5    requires n_times_slot(f, n, inv)
6    requires inv == get_n_times_inv()
7    ensures n_times_inv(s, n)
8  {
9    var i: Int
10   i := 0
11
12   while (i < n)
13     invariant 0 <= i && i <= n
14     invariant n_times_inv(s, i)
15   {
16     n_times_slot_apply_0(f, n, inv, s, i, f, s)
17     i := i + 1
18   }
19 }
```

FIGURE 4.6: The viper encoding of the `n_times` method.

4.4.4 Method Clients

Whenever we call a method with a parametric assertion, we instantiate the parametric assertion either with a predicate or with another parametric assertion. So we know the instantiation of the parametric assertion at the call site. To encode this knowledge, we simply generate a new abstract method for each call site of a method with parametric assertion. Figure 4.7 shows the generated method for a call site where we instantiate `inv` with the predicate `count_inv`.

On line 5 we add the knowledge that `inv` is instantiated with the predicate `count_inv` instead. With this knowledge, we can replace all evaluations of `inv` with evaluating `count_inv` instead. This happened on lines 3 and 6.

Parametric assertions can occur in call slots. In section 3.4.2, we said that we generate a pure boolean function and a helper method or function for each call slot. Call slots with parametric assertions will then be translated as follows. We will only generate the pure boolean function representing the call slot instance. For the helper

```
1 method n_times_0(f: Function, n: Int, s: Ref, inv: Assertion)
2   requires 0 <= n
3   requires count_inv(s, 0)
4   requires n_times_slot(f, n, inv)
5   requires inv == get_count_inv()
6   ensures count_inv(s, n)
```

FIGURE 4.7: The generated method for the call on line 37 of Figure 4.3.

method we will generate a new abstract method for each call site as described above based on the original helper method. The generation of the pure boolean function stays the same.

4.4.5 Call Slot Proofs

Figure 4.8 shows a proof for a call slot that has a parametric assertion. The parametric assertion `inv` is instantiated with the predicate `count_inv`. Inside the proof we want to be able to use this knowledge. Figure 4.9 shows the Viper encoding of that proof. Some aspects of the proof that are independent of parametric assertions have been simplified.

```
1     # ...
2     @CallSlotProof(n_times_slot(f, 23, count_inv))
3     def n_times_slot(f: f_type, n: int, inv: inv_type) -> None:
4
5         @UniversallyQuantified
6         def uq(s: State, i: int) -> None:
7             # Requires(0 <= i and i < n)
8             # Requires(inv(s, i))
9             Unfold(inv(s, i))
10
11            ClosureCall(f(s), count)
12
13            Fold(inv(s, i + 1))
14            # Ensures(inv(s, i + 1))
15    # ...
```

FIGURE 4.8: A call slot proof using parametric assertions from Figure 4.3.

The parametric assertion `inv` is evaluated on the lines 8, 9, 13 and 14 of Figure 4.8. Because we know that it is instantiated with the predicate `count_inv` we can replace these evaluations with its instantiation `count_inv`. This results in the predicate `count_inv` being used on the lines 10, 12, 17 and 19 resp. of the Viper encoding in Figure 4.9.

```
1   /* ... */
2   var b: Bool
3   while (b) {
4     var uq_s: Ref
5     var uq_i: Int
6     var call_counter: Int
7     call_counter := 0
8
9     inhale 0 <= uq_i && uq_i < 23
10    inhale count_inv(uq_s, uq_i)
11
12    unfold count_inv(uq_s, uq_i)
13
14    count_static_dispatch(f, uq_s)
15    call_counter := call_counter + 1
16
17    fold count_inv(uq_s, uq_i + 1)
18
19    exhale count_inv(uq_s, uq_i + 1)
20    assert call_counter == 1
21  }
22  assume n_times_slot(f, 23, get_count_inv())
23  /* ... */
```

FIGURE 4.9: A Viper encoding of the proof from Figure 4.8.

Chapter 5

Implementation

In this chapter we want to elaborate on the implementation we created. For the implementation we modified Nagini to verify call slots. Impure call slots are fully supported including old expressions. Pure call slots enjoy also great support in this implementation where the only limitation is that pure call slots cannot be used in contracts. For call slot proofs we used the same restrictions proposed in section 3.4.4. Unfortunately, a minor bug causes Nagini to crash when nested proofs are used.

Parametric assertions are not part of this implementation. Nevertheless, we can get the same functionality with a workaround. This workaround requires one to know all call sites of a method with parametric assertions. This is generally not desirable and often unrealistic (e.g., when one writes a library to be used by unknown code). Nevertheless, we can demonstrate the functionality and usefulness of parametric assertions with this workaround. Parametric assertions do not suffer from this problem.

In Appendix A we attached some interesting examples that were successfully verified with our implementation.

Nagini always finished within less than 15s for all the examples we verified (a medium range Desktop PC was used in this case). We consider this to be a reasonable time and it is similar to other examples that do not use closures.

Chapter 6

Evaluation

6.1 Discussion

In chapter 3 we introduced call slots. Call slots solve the problem of verifying closure calls when their contracts are unknown. Call slots split the problem into two subproblems. The first subproblem is to establish that a closure can be called in an environment as described by a call slot instance. This problem can be solved through call slot proofs which results in the knowledge that a call slot instance is valid. This knowledge can then be passed from where the proof is carried out to where the closure call occurs. The second subproblem is to verify that the closure call occurs in an environment as described by the call slot instance. This is done through the `ClosureCall` annotation and using a call slot justification. Of course if it is known to which method the closure points, call slots are not needed and instead we can prove static dispatch as a justification for the `ClosureCall` annotation.

In section 3.3 we showed that call slots fully support old expressions. Old epxressions are very important for method contracts as they allow to express many contracts much more precisely. With call slots that support old expressions, we get very strong support for impure closures (closures that are not side-effect free).

Nagini introduces the distinction between pure and impure methods through the `Pure` decorator. Pure methods have some properties that make them attractive for many use cases. They are side-effect free, deterministic and can be used in contracts. In section 3.2 we introduce the same distinction for call slots. Pure call slots have the same attractive three properties of pure methods. Therefore, we get full support for pure closures.

In section 4.1 we show that there are scenarios where we want assertions to be parametric so that a higher-order function can support different closures better. Parametric assertions are a simple solution to this problem. A minor limitation of parametric assertions is that they do not support two state assertions like postconditions do. Nevertheless, Call slots and parametric assertions work very well together. Call slots together with parametric assertions allow us to verify many more interesting use cases than with only call slots.

With call slots one needs to write more specifications. However, these additional specifications allow us to more precisely specify what a closure is allowed to do. For example the call slot in Figure 3.2 only gives an access permission of $1/2$ to `a`'s `parameter` field. The call slot is used in Figure 3.3 on line 21. At that point we have full access to the `parameter` field. With this call slot a closure gets only an access permission of $1/2$, even though it is called where we have full access to the `parameter` field. So while we have to write additional contracts for call slots, we can also restrict what a closure is allowed to do with these contracts. We believe such an approach fits well into deductive verification.

Declaring call slots and carrying out call slot proofs is rather verbose. For call slots we have to separately declare a Python method, contracts and a closure call. For call slot proofs we additionally have to fill in the proof body. With this one has to write a decent amount of additional code to verify closures. We believe there is room for improvement when it comes to writing call slots, call slot proofs and closure calls in Python. However, it is also not easy to find concise ways to declare these constructs in Python. Moreover, we can express all contracts in valid Python so we get proper syntax highlighting and semantic checks for our annotations.

Our implementation supports many important features of call slots. Old expressions are fully supported. Pure call slots are well supported. Only the usage of pure closures in contracts is unsupported in our implementation. With our implementation we could verify a wide variety of different examples (those examples can be found in Appendix A). This demonstrates how well call slots together with parametric assertions allow us to verify closures.

Overall, we believe call slots together with parametric assertions offer a very good solution to verifying closure calls. We are not aware of any severe limitations of this approach. There may be scenarios that are difficult to verify using our solution. However, we believe for the most of these scenarios it is possible to work around them one way or another. Call slots support many features very well (e.g., pure method and old expressions). They work well with existing features of Nagini. They seem like a very natural way to verify closures. We believe together with parametric assertions they can verify the majority of use cases.

6.2 Comparison with existing Work

In the following we want to compare call slots with parametric assertions to existing work. We will first do the comparison with Dafny [19], then with VeriFast [14] and finally compare our methodology to the Pre/Post model which was briefly introduced in section 2.1.

6.2.1 Dafny

As mentioned in section 2.2, Dafny has excellent support for pure closures. Impure closures do not seem to be supported at all in Dafny. Call slots support impure closures naturally. When it comes to verifying pure closures, we think that Dafny imposes a much smaller annotation overhead (its syntax is a lot more concise). However, we believe both approaches are very general. We believe there are few use cases that are support by Dafny but not by call slots or vice versa.

6.2.2 VeriFast

VeriFast uses predicate families to support closures. Predicate families seem to be the only solution to verify closures. VeriFast's predicate families are indexed by function pointers. While this allows one to verify many interesting use cases, there are also limitations to this approach.

In Figure 6.1 two closures `setup` and `compute` are called in sequence. In VeriFast one could introduce two predicate families `pre` and `post` where `pre` corresponds to the precondition and `post` denotes to the postcondition of a closure. Then on line 17, the predicate `post(setup)(...)` would be established and on line 18 the predicate `pre(compute)(...)` has to hold. However, a verifier cannot prove that

the predicate `pre(compute)(...)` holds on line 18, because it is unknown to which method `compute` points. So the predicate `pre(compute)(...)` is also unknown.

This can be solved with parametric assertions and call slots. First we introduce three parametric assertions `before`, `between` and `after`. Then we require that `before` holds before the closure call to `setup`. The assertion `between` has to hold between the two closure calls and the assertion `after` has to hold after the closure call to `compute`. We continue with declaring two call slot `setup_slot` and `compute_slot`, one for each closure call. The precondition of `setup_slot` will be the parametric assertion `before` and the postcondition will be `between`. The assertion `between` will be the precondition of `compute_slot` and `after` will be the postcondition of `compute_slot`. We can then use appropriate call slot instances to verify both closure calls and the higher-order function `f`. A client of `f` can then prove these call slots with the right instantiations for the parametric assertions.

We believe that call slots with parametric assertions support more use cases than VeriFast's predicate families.

```
1  class Argument:
2
3      def __init__(self, a: int, b: int) -> None:
4          self.a = a  # type: int
5          self.b = b  # type: int
6
7          Ensures(Acc(self.a) and Acc(self.b))
8          Ensures(self.a == a and self.b == b)
9
10
11 F_Type = Callable[[Argument, int], Optional[object]]
12
13
14 def f(setup: F_Type, compute: F_Type, arg: Argument, c: int, d: int) ->
       None:
15
16     setup(arg, c)
17
18
19     compute(arg, d)
```

FIGURE 6.1: An example that cannot be verified with VeriFast's predicate families.

6.2.3 Pre/Post Model

We first mentioned the Pre/Post Model in chapter 2. It is very well explained in [17].

We believe that the Pre/Post Model is as expressive as call slots with parametric assertions. That is, both approaches support the same set of use cases. There are still differences between the two and we believe that our approach is a cleaner solution to the problem.

In [17] it is highlighted that the Pre/Post model can suffer from circular reasoning. Because the pre and post placeholders can be used in assertions, it is possible to use them in a circular way that introduces unsoundness. A simple solution to this is given in [17]. This solution is then refined in [16]. Because call slots do not make use of placeholders, call slots do not suffer from this particular problem. However, call slot proofs can also suffer from circular reasoning. This can be solved by requiring call slot proofs to terminate.

In section 2.1 we briefly explained the Pre/Post model. In particular, one needs to be able to link the pre and post placeholders with concrete contracts when it is known to which method a closure points. It is possible to do that in this setting. However, we did not find a way to do this that supports old expressions and the three properties of pure methods (section 3.2 defines these three properties). It is very hard to make the Pre/Post modle work in this setting due to its requirements.

However, call slots' requirements are easily fulfilled in this setting. To encode call slots one needs to be able to:

- declare call slots.
- proof call slot instances.
- pass the knowledge, that a call slot instance is valid, around.
- annotate closure calls appropriately.

Once these requirements are met, every closure call can be rewritten to a method call with known contracts. We believe often these requirements can be met more easily than the requirements of the Pre/Post model.

Chapter 7

Conclusion

Closures and higher-order functions are attractive features of programming languages. However, they complicate modular reasoning.

In this thesis we have proposed call slots and parametric assertions to modularly verify closures and higher-order functions. Call slots allow us to verify closures even though the contracts of closures are often unknown. Call slots naturally support old expressions and pure closures. Parametric assertions allow us to parametrize methods with assertions. They enable us to verify higher-order methods that aim to be generic w.r.t. the contracts of their closures parameters.

We partially implemented this methodology in Nagini. With this implementation we could verify many interesting examples. Verification finishes within reasonable time. Given these findings, we conclude that call slots together with parametric assertions can verify a wide range of interesting use cases and are an attractive approach for verification of closures.

7.1 Future Work

In the following we would like to point at possible future extensions of this work.

In particular, capturing sticks out as an important problem when it comes to verification of closures. We believe there is potential for clean solutions based on our own considerations and previous work (e.g., [17, 21]).

In chapter 4 we propose parametric assertions but only focus on a simple version of them (see section 4.4.1). We believe there are many enhancements possible that would make closure verification easier.

Otherwise, call slots can be further improved in different aspects such as automatic instantiation of universally quantified variables, automating proofs for call slot instances or simply formalizing the concepts of call slots.

Bibliography

[1] Prateek Agarwal, Yannis Kassios, and Peter Müller. *Adding Closure Support to Chalice*. Master's Thesis. ETHZ, Nov. 2010. URL: https://www.ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Prateek_Agarwal_MA_report.pdf.

[2] Michael Barnett et al. "Boogie: A modular reusable verifier for object-oriented programs". In: *FMCO*. Vol. 5. 4111. Springer. 2005, pp. 364–387.

[3] Mike Barnett, K Rustan M Leino, and Wolfram Schulte. "The Spec# programming system: An overview". In: *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*. Springer. 2004, pp. 49–69.

[4] Martin Berger, Kohei Honda, and Nobuko Yoshida. "A Logical Analysis of Aliasing in Imperative Higher-order Functions". In: *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*. ICFP '05. New York, NY, USA: ACM, 2005, pp. 280–293. ISBN: 978-1-59593-064-4. DOI: 10.1145/1086365.1086401. URL: http://doi.acm.org/10.1145/1086365.1086401.

[5] John Boyland. "Checking Interference with Fractional Permissions". en. In: *SpringerLink*. Springer, Berlin, Heidelberg, June 2003, pp. 55–72. DOI: 10.1007/3-540-44898-5_4. URL: https://link.springer.com/chapter/10.1007/3-540-44898-5_4.

[6] Martin Buchi and Wolfgang Weck. *The Greybox Approach: When Blackbox Specifications Hide Too Much*. Tech. rep. Turku Centre for Computer Science, 1999.

[7] Lilian Burdy et al. "An overview of JML tools and applications". English. In: *International Journal on Software Tools for Technology Transfer; Heidelberg* 7.3 (June 2005), pp. 212–232. ISSN: 14332779. DOI: http://dx.doi.org/10.1007/s10009-004-0167-4. URL: http://search.proquest.com/docview/197472184/abstract/600B03C8F0144A48PQ/1.

[8] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08. Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340. ISBN: 3-540-78799-2, 978-3-540-78799-0. URL: http://dl.acm.org/citation.cfm?id=1792734.1792766.

[9] Alexandru Dima. "Implementing Closures in Dafny". In: (Sept. 2010). URL: http://www.pm.inf.ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Alexandru_Dima_RS_report.pdf.

[10] George W. Ernst, Jainendra K. Navlakha, and William F. Ogden. "Verification of programs with procedure-type parameters". en. In: *Acta Informatica* 18.2 (Nov. 1982), pp. 149–169. ISSN: 0001-5903, 1432-0525. DOI: 10.1007/BF00264436. URL: https://link.springer.com/article/10.1007/BF00264436.

[11] Kohei Honda. "From Process Logic to Program Logic". In: *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*. ICFP '04. New York, NY, USA: ACM, 2004, pp. 163–174. ISBN: 978-1-58113-905-1. DOI: 10.1145/1016850.1016874. URL: http://doi.acm.org/10.1145/1016850.1016874.

[12] Kohei Honda and Nobuko Yoshida. "A Compositional Logic for Polymorphic Higher-order Functions". In: *Proceedings of the 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. PPDP '04. New York, NY, USA: ACM, 2004, pp. 191–202. ISBN: 978-1-58113-819-1. DOI: 10.1145/1013963.1013985. URL: http://doi.acm.org/10.1145/1013963.1013985.

[13] Kohei Honda, Nobuko Yoshida, and Martin Berger. "An observationally complete program logic for imperative higher-order functions". In: *Theoretical Computer Science* 517 (Jan. 2014), pp. 75–101. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2013.11.003. URL: http://www.sciencedirect.com/science/article/pii/S0304397513008219.

[14] Bart Jacobs and Frank Piessens. *The VeriFast Program Verifier*. 2008.

[15] Ioannis T Kassios. "Dynamic frames: Support for framing, dependencies and sharing without restrictions". In: *International Symposium on Formal Methods*. Springer. 2006, pp. 268–283.

[16] Ioannis T. Kassios and Peter Müller. "Modular Specification and Verification of Delegation with SMT Solvers". In: (2011).

[17] Ioannis T. Kassios and Peter Müller. "Specification and Verification of Closures". In: (2010).

[18] Neelakantan R. Krishnaswami et al. "Design patterns in separation logic". In: *Proceedings of the 4th international workshop on Types in language design and implementation*. ACM, 2009, pp. 105–116. URL: http://dl.acm.org/citation.cfm?id=1481874.

[19] K. Rustan M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". en. In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer, Berlin, Heidelberg, Apr. 2010, pp. 348–370. DOI: 10.1007/978-3-642-17511-4_20. URL: https://link.springer.com/chapter/10.1007/978-3-642-17511-4_20.

[20] K Rustan M Leino, Peter Müller, and Jan Smans. "Verification of Concurrent Programs with Chalice." In: *FOSAD*. Vol. 5705. Springer. 2009, pp. 195–222.

[21] F. Meier. "Closure verification in an automated fractional permission setting". In: (2014). URL: https://www.ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Fabian_Meier_BA_report.pdf.

[22] Bertrand Meyer. *Eiffel: The Language*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1992. ISBN: 0-13-247925-7.

[23] P. Müller, M. Schwerhoff, and A. J. Summers. "Viper: A Verification Infrastructure for Permission-Based Reasoning". In: *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Ed. by B. Jobstmann and K. R. M. Leino. Vol. 9583. LNCS. Springer-Verlag, 2016, pp. 41–62.

[24] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. "Hoare type theory, polymorphism and separation". In: *Journal of Functional Programming* 18.5-6 (Sept. 2008), pp. 865–911. ISSN: 1469-7653, 0956-7968. DOI: 10.1017/S0956796808006953. URL: https://www.cambridge.org/core/journals/journal-of-functional-programming/article/hoare-type-theory-polymorphism-and-separation1/D6B10CE5025B4C895C2FC7438393195E.

[25] Martin Nordio et al. "Reasoning about Function Objects". In: (Feb. 2009). URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.182.9110.

[26] Steve M. Shaner, Gary T. Leavens, and David A. Naumann. "Modular Verification of Higher-order Methods with Mandatory Calls Specified by Model Programs". In: *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*. OOPSLA '07. New York, NY, USA: ACM, 2007, pp. 351–368. ISBN: 978-1-59593-786-5. DOI: 10.1145/1297027.1297053. URL: http://doi.acm.org/10.1145/1297027.1297053.

[27] Nobuko Yoshida, Kohei Honda, and Martin Berger. "Logical Reasoning for Higher-Order Functions with Local State". en. In: *Foundations of Software Science and Computational Structures*. Springer, Berlin, Heidelberg, Mar. 2007, pp. 361–377. DOI: 10.1007/978-3-540-71389-0_26. URL: https://link.springer.com/chapter/10.1007/978-3-540-71389-0_26.

Appendix A

Examples Successfully Verified by the Implementation

```
from typing import Callable, Tuple
from nagini_contracts.contracts import (
    Requires,
    Ensures,
    Invariant,
    Acc,
    Result,
    Pure,
    CallSlot,
    CallSlotProof,
    UniversallyQuantified,
    ClosureCall,
)


@Pure
def idiv(x: int, y: int) -> int:
    Requires(y != 0)


@Pure
def sqrt(x: int) -> int:
    Requires(x >= 0)


@Pure
def log(x: int) -> int:
    Requires(x > 0)


def choice() -> bool:
    return True


class Argument:

    def __init__(self, parameter: int, result: int) -> None:
        self.parameter = parameter  # type: int
        self.result = result  # type: int

        Ensures(Acc(self.parameter) and Acc(self.result))
        Ensures(self.parameter == parameter and self.result == result)


G_Type = Callable[[Argument, int, int], Tuple[int, int]]

```

```
47
48 @CallSlot
49 def func_call_slot(g: G_Type, b: int, c: int) -> None:
50
51     @UniversallyQuantified
52     def uq(a: Argument) -> None:
53
54         Requires(Acc(a.parameter, 1 / 2) and Acc(a.result))
55         Requires(a.parameter >= b + c)
56
57         ret = g(a, b, c)
58
59         Ensures(Acc(a.parameter, 1 / 2) and Acc(a.result))
60         Ensures(a.result != 0 and ret[0] >= 0 and ret[1] > 0)
61
62
63 def func(
64         g: G_Type,
65         a: Argument,
66         b: int,
67         c: int
68 ) -> Tuple[int, int, int]:
69
70     Requires(func_call_slot(g, b, c))
71     Requires(Acc(a.parameter))
72     Requires(Acc(a.result))
73     Ensures(Acc(a.parameter))
74     Ensures(Acc(a.result))
75
76     while a.parameter < b + c:
77         Invariant(Acc(a.parameter))
78         a.parameter = a.parameter * a.parameter
79
80     # g reads a.parameter and writes to a.result
81
82     # closure call justified because the call slot holds:
83     # func_call_slot(g, b, c)
84     d, e = ClosureCall(g(a, b, c), func_call_slot(g, b, c)(a))  # type:
       Tuple[int, int]
85
86     return idiv(1, a.result), sqrt(d), log(e)
87
88
89 def concrete_g_1(a: Argument, b: int, c: int) -> Tuple[int, int]:
90     Requires(Acc(a.parameter, 1 / 2) and Acc(a.result))
91     Requires(a.parameter >= b + c)
92     Ensures(Acc(a.parameter, 1 / 2) and Acc(a.result))
93     Ensures(a.result == -1 and Result()[0] == 0 and Result()[1] >= 1)
94
95     a.result = -1
96     return 0, a.parameter - b - c + 1
97
98
99 def concrete_g_2(a: Argument, b: int, c: int) -> Tuple[int, int]:
100     Requires(Acc(a.parameter, 1 / 2) and Acc(a.result))
101     Requires(a.parameter >= b + c)
102     Ensures(Acc(a.parameter, 1 / 2) and Acc(a.result))
103     Ensures(a.result == 1 and Result()[0] >= 0 and Result()[1] == 1)
104
105     a.result = 1
106     return a.parameter - b - c, 1
107
108
```

```
109 def client() -> None:
110
111     if choice():  # non-deterministic choice
112         concrete_g = concrete_g_1
113     else:
114         concrete_g = concrete_g_2
115
116     a = Argument(2, 2)
117     b, c = 2, 3
118
119     @CallSlotProof(func_call_slot(concrete_g, b, c))
120     def func_call_slot_proof(concrete_g: G_Type, b: int, c: int) -> None:
121
122         @UniversallyQuantified
123         def uq(a: Argument) -> None:
124
125             # Requires(Acc(a.parameter, 1 / 2) and Acc(a.result))
126             # Requires(a.parameter >= b + c)
127
128             if concrete_g == concrete_g_1:
129                 # closure call justified, because we can prove static
                        dispatch:
130                 # concrete_g == concrete_g_1
131                 # and concrete_g_1 is a method whose contracts we can look
                        up
132                 # statically/in nagini
133                 ret = ClosureCall(concrete_g(a, b, c), concrete_g_1)  #
                        type: Tuple[int, int]
134             else:
135                 assert concrete_g == concrete_g_2
136                 # closure call justified, because we can prove static
                        dispatch:
137                 # concrete_g == concrete_g_2
138                 # and concrete_g_2 is a method whose contracts we can look
                        up
139                 # statically/in nagini
140                 ret = ClosureCall(concrete_g(a, b, c), concrete_g_2)
141
142             # Ensures(Acc(a.parameter, 1 / 2) and Acc(a.result))
143             # Ensures(a.result != 0 and ret[0] >= 0 and ret[1] > 0)
144
145     func(concrete_g, a, b, c)
```

LISTING A.1: Static Code Example

```
1  from typing import Callable, Optional
2  from nagini_contracts.contracts import (
3      Requires,
4      Ensures,
5      Acc,
6      Predicate,
7      Fold,
8      Unfold,
9      CallSlot,
10     CallSlotProof,
11     UniversallyQuantified,
12     ClosureCall,
13 )
14
15
16 class Argument:
17
18     def __init__(self, a: int, b: int) -> None:
```

```
19          self.a = a  # type: int
20          self.b = b  # type: int
21
22          Ensures(Acc(self.a) and Acc(self.b))
23          Ensures(self.a == a and self.b == b)
24
25
26  F_Type = Callable[[Argument, int], Optional[object]]
27
28
29  @CallSlot
30  def f_setup(setup: F_Type, c: int, d: int, before_token: int,
        between_token: int) -> None:
31
32      @UniversallyQuantified
33      def uq(arg: Argument) -> None:
34          Requires(before(arg, c, d, before_token))
35
36          setup(arg, c)
37
38          Ensures(between(arg, c, d, between_token))
39
40
41  @CallSlot
42  def f_compute(compute: F_Type, c: int, d: int, between_token: int,
        after_token: int) -> None:
43
44      @UniversallyQuantified
45      def uq(arg: Argument) -> None:
46          Requires(between(arg, c, d, between_token))
47
48          compute(arg, d)
49
50          Ensures(after(arg, c, d, after_token))
51
52
53  def f(
54          setup: F_Type,
55          compute: F_Type,
56          arg: Argument,
57          c: int,
58          d: int,
59          before_token: int,
60          between_token: int,
61          after_token: int
62  ) -> None:
63
64      Requires(f_setup(setup, c, d, before_token, between_token))
65      Requires(f_compute(compute, c, d, between_token, after_token))
66      Requires(before(arg, c, d, before_token))
67
68      ClosureCall(setup(arg, c), f_setup(setup, c, d, before_token,
            between_token)(arg))
69
70      # assert between(arg, c, d, between_token)
71
72      ClosureCall(compute(arg, d), f_compute(compute, c, d, between_token,
            after_token)(arg))
73
74      Ensures(after(arg, c, d, after_token))
75
76
77  def setup(arg: Argument, c: int) -> Optional[object]:
```

```
78      Requires(Acc(arg.a))
79      Requires(c > 1)
80      Ensures(Acc(arg.a))
81      Ensures(arg.a == 3 * c)
82
83      arg.a = 3 * c
84
85
86  def compute(arg: Argument, d: int) -> Optional[object]:
87      Requires(Acc(arg.a, 1 / 2) and Acc(arg.b))
88      Ensures(Acc(arg.a, 1 / 2) and Acc(arg.b))
89      Ensures(arg.b == arg.a + d)
90
91      arg.b = arg.a + d
92
93
94  @Predicate
95  def before(arg: Argument, c: int, d: int, token: int) -> bool:
96      return (
97          Acc(arg.a) and Acc(arg.b) and c > 1 if token == 1 else
98          True
99      )
100
101
102 @Predicate
103 def between(arg: Argument, c: int, d: int, token: int) -> bool:
104     return (
105         Acc(arg.b) and Acc(arg.a) and arg.a == 3 * c if token == 1 else
106         True
107     )
108
109
110 @Predicate
111 def after(arg: Argument, c: int, d: int, token: int) -> bool:
112     return (
113         Acc(arg.a) and Acc(arg.b) and arg.a == 3 * c and arg.b == arg.a +
                d if token == 1 else
114         True
115     )
116
117
118 def client() -> None:
119
120     arg = Argument(2, 2)
121     c, d = 2, 3
122     before_token = between_token = after_token = 1
123
124     @CallSlotProof(f_setup(setup, c, d, before_token, between_token))
125     def f_setup_proof(f: F_Type, c: int, d: int, A: int, B: int) -> None:
126
127         @UniversallyQuantified
128         def uq(arg: Argument) -> None:
129             # Requires(before(arg, c, d, before_token))
130             Unfold(before(arg, c, d, between_token))
131
132             ClosureCall(f(arg, c), setup)
133
134             Fold(between(arg, c, d, between_token))
135             # Ensures(between(arg, c, d, between_token))
136
137     @CallSlotProof(f_compute(compute, c, d, between_token, after_token))
138     def f_compute_proof(f: F_Type, c: int, d: int, B: int, C: int) -> None
            :
```

```
139
140         @UniversallyQuantified
141         def uq(arg: Argument) -> None:
142             # Requires(between(arg, c, d, between_token))
143             Unfold(between(arg, c, d, between_token))
144
145             ClosureCall(f(arg, d), compute)
146
147             Fold(after(arg, c, d, after_token))
148             # Ensures(after(arg, c, d, after_token))
149
150     Fold(before(arg, c, d, between_token))
151     f(setup, compute, arg, c, d, before_token, between_token, after_token)
152     Unfold(after(arg, c, d, after_token))
153
154     assert arg.a == 6
155     assert arg.b == 9
```

LISTING A.2: Setup-Compute Example

```
1  from typing import Callable, Optional
2  from nagini_contracts.contracts import (
3      Implies,
4      Requires,
5      Ensures,
6      Result,
7      Old,
8      Invariant,
9      Predicate,
10     Fold,
11     Unfold,
12     Unfolding,
13     Pure,
14     Acc,
15     CallSlot,
16     CallSlotProof,
17     UniversallyQuantified,
18     ClosureCall,
19     Assert,
20 )
21
22
23 class State:
24
25     def __init__(
26         self,
27         counter: int,
28         value: int,
29         offset: int,
30         next: Optional['State']
31     ) -> None:
32
33         self.counter = counter
34         self.value = value
35         self.offset = offset
36         self.next = next
37         Ensures(Acc(self.counter) and Acc(self.value))
38         Ensures(Acc(self.offset) and Acc(self.next))
39         Ensures(self.counter == counter and self.value == value)
40         Ensures(self.offset == offset and self.next == next)
41
42
43 f_type = Callable[[State], Optional[object]]
```

```
44
45
46  @Predicate
47  def n_inv(s: State, i: int, n_inv_token: int) -> bool:
48      return (
49              Acc(s.counter) and Acc(s.value) and
50              s.counter == i and s.value == i * (i + 1) // 2
51          if n_inv_token == 1 else
52          True
53      )
54
55
56  @CallSlot
57  def n_times_slot(f: f_type, n_inv_token: int) -> None:
58
59      @UniversallyQuantified
60      def uq(s: State, i: int) -> None:
61          Requires(n_inv(s, i, n_inv_token))
62
63          f(s)
64
65          Ensures(n_inv(s, i + 1, n_inv_token))
66
67
68  def n_times(f: f_type, n: int, s: State, n_inv_token: int) -> None:
69      Requires(0 <= n)
70      Requires(n_inv(s, 0, n_inv_token))
71      Requires(n_times_slot(f, n_inv_token))
72      Ensures(n_inv(s, n, n_inv_token))
73
74      i = 0
75      while i < n:
76          Invariant(0 <= i and i <= n)
77          Invariant(n_inv(s, i, n_inv_token))
78
79          ClosureCall(f(s), n_times_slot(f, n_inv_token)(s, i))
80          i += 1
81
82      # This shouldn't be necessary
83      # Seems to be a bug in Nagini
84      Unfold(n_inv(s, i, n_inv_token))
85      Fold(n_inv(s, n, n_inv_token))
86
87
88  def sum_range(s: State) -> Optional[object]:
89      Requires(Acc(s.counter) and Acc(s.value))
90      Ensures(Acc(s.counter) and Acc(s.value))
91      Ensures(s.counter == Old(s.counter) + 1)
92      Ensures(s.value == Old(s.value + s.counter + 1))
93
94      s.counter += 1
95      s.value += s.counter
96
97      return None
98
99
100 def n_times_client() -> None:
101
102     s = State(0, 0, 0, None)
103
104     f = sum_range
105
106     @CallSlotProof(n_times_slot(f, 1))
```

```
    def n_times_slot(f: f_type, n_inv_token: int) -> None:

        @UniversallyQuantified
        def uq(s: State, i: int) -> None:
            # Requires(n_inv(s, i, n_inv_token))
            Unfold(n_inv(s, i, n_inv_token))

            assert i == s.counter
            assert s.value == i * (i + 1) // 2
            ClosureCall(f(s), sum_range)

            assert s.counter == i + 1
            assert s.value == i * (i + 1) // 2 + i + 1
            assert s.value == i * (i + 1) // 2 + i + 1
            assert s.value == i * (i + 1) // 2 + 2 * (i + 1) // 2
            assert s.value == (i * (i + 1) + 2 * (i + 1)) // 2

            Fold(n_inv(s, i + 1, n_inv_token))
            # Ensures(n_inv(s, i + 1, n_inv_token))

    Fold(n_inv(s, 0, 1))
    n_times(f, 23, s, 1)
    Unfold(n_inv(s, 23, 1))
    assert s.value == 276
    assert s.counter == 23
```

LISTING A.3: N-Times Example

```
from typing import Callable, Optional
from nagini_contracts.contracts import (
    Implies,
    Requires,
    Ensures,
    Result,
    Old,
    Invariant,
    Predicate,
    Fold,
    Unfold,
    Unfolding,
    Pure,
    Acc,
    CallSlot,
    CallSlotProof,
    UniversallyQuantified,
    ClosureCall,
    Assert,
)


class State:

    def __init__(
        self,
        counter: int,
        value: int,
        offset: int,
        next: Optional['State']
    ) -> None:

        self.counter = counter
        self.value = value
        self.offset = offset
```

```
36          self.next = next
37          Ensures(Acc(self.counter) and Acc(self.value))
38          Ensures(Acc(self.offset) and Acc(self.next))
39          Ensures(self.counter == counter and self.value == value)
40          Ensures(self.offset == offset and self.next == next)
41
42
43  cond_t = Callable[[State], bool]
44  body_t = Callable[[State], Optional[object]]
45
46
47  @Predicate
48  def inv(s: State, inv_token: int) -> bool:
49      return (
50          (
51              Acc(s.counter) and Acc(s.value, 1 / 2) and
52              s.counter <= s.value
53          ) if inv_token == 1 else
54          True
55      )
56
57
58  @Pure
59  def cond_expr(s: State, inv_token: int) -> bool:
60      Requires(inv(s, inv_token))
61      return Unfolding(
62          inv(s, inv_token),
63          s.counter < s.value if inv_token == 1 else
64          True
65      )
66
67
68  @Pure
69  @CallSlot
70  def cond_slot(
71      cond: cond_t,
72      inv_token: int
73  ) -> None:
74
75      @UniversallyQuantified
76      def uq(s: State) -> None:
77          Requires(inv(s, inv_token))
78
79          b = cond(s)
80
81          Ensures(b == cond_expr(s, inv_token))
82
83
84  @CallSlot
85  def body_slot(body: body_t, inv_token: int) -> None:
86
87      @UniversallyQuantified
88      def uq(s: State) -> None:
89          Requires(inv(s, inv_token))
90          Requires(cond_expr(s, inv_token))
91
92          body(s)
93
94          Ensures(inv(s, inv_token))
95
96
97  def while_loop(
98      cond: cond_t,
```

```
    body: body_t,
    s: State,
    inv_token: int
) -> None:

    Requires(inv(s, inv_token))
    Requires(cond_slot(cond, inv_token))
    Requires(body_slot(body, inv_token))

    Ensures(inv(s, inv_token))
    Ensures(not cond_expr(s, inv_token))

    b = ClosureCall(
        cond(s),
        cond_slot(cond, inv_token)(s)
    )  # type: bool

    assert b == cond_expr(s, inv_token)

    while b:
        Invariant(inv(s, inv_token))
        Invariant(b == cond_expr(s, inv_token))

        ClosureCall(
            body(s),
            body_slot(body, inv_token)(s)
        )

        b = ClosureCall(
            cond(s),
            cond_slot(cond, inv_token)(s)
        )


@Pure
def count_to_cond(s: State) -> bool:
    Requires(Acc(s.counter) and Acc(s.value))
    Ensures(Result() == (s.counter < s.value))
    return s.counter < s.value


def count_to_body(s: State) -> Optional[object]:
    Requires(Acc(s.counter) and Acc(s.value, 1 / 2))
    Requires(s.counter < s.value)

    Ensures(Acc(s.counter) and Acc(s.value, 1 / 2))
    Ensures(s.counter <= s.value)
    Ensures(s.counter == Old(s.counter) + 1)

    s.counter += 1
    return None


def while_loop_client() -> None:

    cond_f = count_to_cond
    body_f = count_to_body

    s = State(0, 20, 0, None)

    @CallSlotProof(cond_slot(cond_f, 1))
    def cond_slot(
        cond: cond_t,
```

```
162         inv_token: int
163     ) -> None:
164 
165         @UniversallyQuantified
166         def uq(s: State) -> None:
167             # Requires(inv(s, inv_token))
168 
169             Unfold(inv(s, inv_token))
170             b = ClosureCall(cond(s), count_to_cond)  # type: bool
171             Fold(inv(s, inv_token))
172 
173             # Ensures(b == cond_expr(s, inv_token))
174 
175     @CallSlotProof(body_slot(body_f, 1))
176     def body_slot(body: body_t, inv_token: int) -> None:
177 
178         @UniversallyQuantified
179         def uq(s: State) -> None:
180             # Requires(inv(s, inv_token))
181             # Requires(Unfolding(inv(s, inv_token), cond_expr(s, inv_token
                    )))
182 
183             Unfold(inv(s, inv_token))
184             ClosureCall(body(s), count_to_body)
185             Fold(inv(s, inv_token))
186 
187             # Ensures(inv(s, inv_token))
188 
189     Fold(inv(s, 1))
190     while_loop(cond_f, body_f, s, 1)
191     Unfold(inv(s, 1))
192 
193     assert s.counter == 20 and s.value == 20
```

LISTING A.4: While Loop Example

```
1  from typing import Callable
2  from nagini_contracts.contracts import (
3      Implies,
4      Requires,
5      Ensures,
6      Result,
7      Old,
8      Invariant,
9      Predicate,
10     Fold,
11     Unfold,
12     Unfolding,
13     Acc,
14     CallSlot,
15     CallSlotProof,
16     UniversallyQuantified,
17     ClosureCall,
18 )
19 
20 
21 def inc(x: int) -> int:
22     Ensures(Result() == x + 1)
23     return x + 1
24 
25 
26 def mul(x: int) -> int:
27     Requires(x > 0)
```

```
28      Ensures(Result() == x * 2)
29      return x * 2
30
31
32  @Predicate
33  def pre(x: int, pre_token: int) -> bool:
34      return (
35          True if pre_token == 1 else
36          x > 0 if pre_token == 2 else
37          True
38      )
39
40
41  @Predicate
42  def post(x: int, ret: int, post_token: int) -> bool:
43      return (
44          ret == x + 1 if post_token == 1 else
45          ret == x * 2 if post_token == 2 else
46          True
47      )
48
49
50  f0_type = Callable[[int], int]
51
52
53  @CallSlot
54  def f1_slot(f0: f0_type, x: int, pre_token: int, post_token: int) -> None:
55      Requires(pre(x, pre_token))
56      ret = f0(x)
57      Ensures(post(x, ret, post_token))
58
59
60  def f1(f0: f0_type, x: int, pre_token: int, post_token: int) -> int:
61      Requires(f1_slot(f0, x, pre_token, post_token))
62      Requires(pre(x, pre_token))
63      Ensures(post(x, Result(), post_token))
64      return ClosureCall(f0(x), f1_slot(f0, x, pre_token, post_token)())
65
66
67  f1_type = Callable[[f0_type, int, int, int], int]
68
69
70  @CallSlot
71  def f2_slot(f1: f1_type, f0: f0_type, x: int, pre_token: int, post_token:
        int) -> None:
72      Requires(pre(x, pre_token))
73      ret = f1(f0, x, pre_token, post_token)
74      Ensures(post(x, ret, post_token))
75
76
77  def f2(f1: f1_type, f0: f0_type, x: int, pre_token: int, post_token: int)
        -> int:
78      Requires(f2_slot(f1, f0, x, pre_token, post_token))
79      Requires(pre(x, pre_token))
80      Ensures(post(x, Result(), post_token))
81      return ClosureCall(
82          f1(f0, x, pre_token, post_token),
83          f2_slot(f1, f0, x, pre_token, post_token)()
84      )
85
86
87  f2_type = Callable[[f1_type, f0_type, int, int, int], int]
88
```

```
89
90  @CallSlot
91  def f3_slot(f2: f2_type, f1: f1_type, f0: f0_type, x: int, pre_token: int,
        post_token: int) -> None:
92      Requires(pre(x, pre_token))
93      ret = f2(f1, f0, x, pre_token, post_token)
94      Ensures(post(x, ret, post_token))
95
96
97  def f3(f2: f2_type, f1: f1_type, f0: f0_type, x: int, pre_token: int,
        post_token: int) -> int:
98      Requires(f3_slot(f2, f1, f0, x, pre_token, post_token))
99      Requires(pre(x, pre_token))
100     Ensures(post(x, Result(), post_token))
101     return ClosureCall(
102         f2(f1, f0, x, pre_token, post_token),
103         f3_slot(f2, f1, f0, x, pre_token, post_token)()
104     )
105
106
107 def client() -> None:
108
109     _inc = inc
110     _mul = mul
111
112     _f1 = f1
113     _f2 = f2
114     _f3 = f3
115
116     @CallSlotProof(f1_slot(_inc, 5, 1, 1))
117     def f1_slot_inc(f0: f0_type, x: int, pre_token: int, post_token: int)
            -> None:
118         # Requires(pre(x, pre_token))
119         Unfold(pre(x, pre_token))
120         ret = ClosureCall(f0(x), inc)  # type: int
121         Fold(post(x, ret, post_token))
122         # Ensures(post(x, ret, post_token))
123
124     @CallSlotProof(f2_slot(_f1, _inc, 5, 1, 1))
125     def f2_slot_inc(_f1: f1_type, _f0: f0_type, x: int, pre_token: int,
            post_token: int) -> None:
126         # Requires(pre(x, pre_token))
127
128         ret = ClosureCall(_f1(_f0, x, pre_token, post_token), f1)  # type:
                int
129
130         # Ensures(post(x, ret, post_token))
131
132     @CallSlotProof(f3_slot(_f2, _f1, _inc, 5, 1, 1))
133     def f3_slot_inc(_f2: f2_type, _f1: f1_type, _f0: f0_type, x: int,
            pre_token: int, post_token: int) -> None:
134         # Requires(pre(x, pre_token))
135
136         ret = ClosureCall(_f2(_f1, _f0, x, pre_token, post_token), f2)  #
                type: int
137
138         # Ensures(post(x, ret, post_token))
139
140     Fold(pre(5, 1))
141     y1 = ClosureCall(_f3(_f2, _f1, _inc, 5, 1, 1), f3)  # type: int
142     Unfold(post(5, y1, 1))
143     assert y1 == 6
144
```

```
145     @CallSlotProof(f1_slot(_mul, 5, 2, 2))
146     def f1_slot_mul(f0: f0_type, x: int, pre_token: int, post_token: int)
            -> None:
147         # Requires(pre(x, pre_token))
148         Unfold(pre(x, pre_token))
149         ret = ClosureCall(f0(x), mul)  # type: int
150         Fold(post(x, ret, post_token))
151         # Ensures(post(x, ret, post_token))
152
153     @CallSlotProof(f2_slot(_f1, _mul, 5, 2, 2))
154     def f2_slot_mul(_f1: f1_type, _f0: f0_type, x: int, pre_token: int,
            post_token: int) -> None:
155         # Requires(pre(x, pre_token))
156
157         ret = ClosureCall(_f1(_f0, x, pre_token, post_token), f1)  # type:
                int
158
159         # Ensures(post(x, ret, post_token))
160
161     @CallSlotProof(f3_slot(_f2, _f1, _mul, 5, 2, 2))
162     def f3_slot_mul(_f2: f2_type, _f1: f1_type, _f0: f0_type, x: int,
            pre_token: int, post_token: int) -> None:
163         # Requires(pre(x, pre_token))
164
165         ret = ClosureCall(_f2(_f1, _f0, x, pre_token, post_token), f2)  #
                type: int
166
167         # Ensures(post(x, ret, post_token))
168
169     Fold(pre(5, 2))
170     y2 = ClosureCall(_f3(_f2, _f1, _mul, 5, 2, 2), f3)  # type: int
171     Unfold(post(5, y2, 2))
172     assert y2 == 10
```

LISTING A.5: HOF Forwarding Example

```
1  from typing import Callable, Optional
2  from nagini_contracts.contracts import (
3      Requires,
4      Ensures,
5      Predicate,
6      Acc,
7      Old,
8      CallSlot,
9      CallSlotProof,
10     ClosureCall,
11 )
12
13
14 class Argument:
15
16     def __init__(self, parameter: int, result: int) -> None:
17         self.parameter = parameter  # type: int
18         self.result = result  # type: int
19
20         Ensures(Acc(self.parameter) and Acc(self.result))
21         Ensures(self.parameter == parameter and self.result == result)
22
23
24 inc_type = Callable[[Argument], Optional[object]]
25
26
27 def inc(arg: Argument) -> Optional[object]:
```

```
    Requires(Acc(arg.parameter) and Acc(arg.result))

    Ensures(Acc(arg.parameter) and Acc(arg.result))
    Ensures(arg.result == Old(arg.result) + arg.parameter)
    Ensures(arg.parameter == Old(arg.parameter))

    arg.result = arg.result + arg.parameter

    return None


@CallSlot
def inc_call_slot(f: inc_type, arg: Argument) -> None:
    Requires(Acc(arg.parameter) and Acc(arg.result))

    f(arg)

    Ensures(Acc(arg.parameter) and Acc(arg.result))
    Ensures(arg.result >= Old(arg.result) + arg.parameter)
    Ensures(arg.parameter == Old(arg.parameter))


def test() -> None:

    arg = Argument(1, 2)

    arg.result = 20
    arg.parameter = 50

    f = inc

    @CallSlotProof(inc_call_slot(inc, arg))
    def inc_proof(f: inc_type, arg: Argument) -> None:
        # Requires(Acc(arg.parameter) and Acc(arg.result))

        ClosureCall(f(arg), inc)

        # Ensures(Acc(arg.parameter) and Acc(arg.result))
        # Ensures(arg.result >= Old(arg.result) + arg.parameter)
        # Ensures(arg.parameter == Old(arg.parameter))

    ClosureCall(f(arg), inc_call_slot(f, arg)())

    assert arg.result >= 70 and arg.parameter == 50
```

LISTING A.6: Old Expressions Example

```
from typing import Callable
from nagini_contracts.contracts import (
    Requires,
    Ensures,
    Acc,
    Pure,
    Result,
    Old,
    CallSlot,
    CallSlotProof,
    UniversallyQuantified,
    ClosureCall
)


def choice() -> bool:
```

```
17      return True
18
19
20  class Argument:
21
22      def __init__(self, parameter: int, result: int) -> None:
23          self.parameter = parameter  # type: int
24          self.result = result  # type: int
25
26          Ensures(Acc(self.parameter) and Acc(self.result))
27          Ensures(self.parameter == parameter and self.result == result)
28
29
30  F_Type = Callable[[Argument, int], int]
31
32
33  @Pure
34  def add(arg: Argument, x: int) -> int:
35      Requires(Acc(arg.parameter))
36      Ensures(Result() == x + arg.parameter)
37      return x + arg.parameter
38
39
40  @Pure
41  def mul(arg: Argument, x: int) -> int:
42      Requires(Acc(arg.parameter))
43      Ensures(Result() == x * arg.parameter)
44      return x * arg.parameter
45
46
47  @Pure
48  @CallSlot
49  def pure_call_slot(f: F_Type, arg: Argument) -> None:
50
51      @UniversallyQuantified
52      def uq(x: int) -> None:
53          Requires(Acc(arg.parameter) and arg.parameter > 0 and x > 1)
54
55          y = f(arg, x)
56
57          Ensures(y > arg.parameter)
58
59
60  def client(f: F_Type, arg: Argument) -> None:
61      Requires(Acc(arg.parameter) and Acc(arg.result))
62      Requires(arg.parameter > 0)
63      Requires(pure_call_slot(f, arg))
64      Ensures(Acc(arg.parameter) and Acc(arg.result))
65      Ensures(arg.parameter == Old(arg.parameter))
66      Ensures(arg.result > arg.parameter)
67
68      arg.result = ClosureCall(f(arg, 20), pure_call_slot(f, arg)(20))
69
70
71  def method() -> None:
72
73      if choice():
74          f = add
75      else:
76          f = mul
77
78      arg = Argument(10, 5)
79
```

```
80      @CallSlotProof(pure_call_slot(f, arg))
81      def pure_call_slot(f: F_Type, arg: Argument) -> None:
82
83          @UniversallyQuantified
84          def uq(x: int) -> None:
85              # Requires(Acc(arg.parameter) and arg.parameter > 0 and x > 1)
86
87              if f == add:
88                  y = ClosureCall(f(arg, x), add)  # type: int
89              else:
90                  y = ClosureCall(f(arg, x), mul)
91
92              # Ensures(y > arg.parameter)
93
94      client(f, arg)
95
96      assert arg.parameter == 10
97      assert arg.result > arg.parameter
```

LISTING A.7: Pure Example 1

```
1   from typing import Callable
2   from nagini_contracts.contracts import (
3       CallSlot,
4       CallSlotProof,
5       UniversallyQuantified,
6       ClosureCall,
7       Pure,
8       Result,
9       Requires,
10      Ensures,
11      Acc
12  )
13
14
15  def choice() -> bool:
16      return True
17
18
19  class Argument:
20
21      def __init__(self, value_a: int, value_b: int) -> None:
22          self.value_a = value_a
23          self.value_b = value_b
24          Ensures(Acc(self.value_a) and Acc(self.value_b))
25          Ensures(self.value_a == value_a and self.value_b == value_b)
26
27
28  f_type = Callable[[Argument, int], int]
29
30
31  @Pure
32  def add(arg: Argument, x: int) -> int:
33      Requires(Acc(arg.value_a))
34      Ensures(Result() == arg.value_a + x)
35      return arg.value_a + x
36
37
38  @Pure
39  def mul(arg: Argument, x: int) -> int:
40      Requires(Acc(arg.value_b))
41      Ensures(Result() == arg.value_b * x)
42      return arg.value_b * x
```

```
43
44
45  @Pure
46  @CallSlot
47  def add_or_mul(f: f_type) -> None:
48
49      @UniversallyQuantified
50      def uq(arg: Argument, x: int) -> None:
51          Requires(Acc(arg.value_a) and Acc(arg.value_b))
52
53          y = f(arg, x)
54
55          Ensures(y == arg.value_a + x or y == arg.value_b * x)
56
57
58  @CallSlot
59  def hof_slot(f: f_type) -> None:
60
61      @UniversallyQuantified
62      def uq(arg: Argument, x: int) -> None:
63          Requires(Acc(arg.value_a, 1 / 2) and Acc(arg.value_b, 1 / 3))
64
65          y = f(arg, x)
66
67          Ensures(Acc(arg.value_a, 1 / 2) and Acc(arg.value_b, 1 / 3))
68          Ensures(y <= arg.value_a + x or y >= arg.value_b * x)
69
70
71  def hof(f: f_type, arg: Argument) -> int:
72      Requires(Acc(arg.value_a, 1 / 2) and Acc(arg.value_b, 1 / 3))
73      Requires(hof_slot(f))
74
75      Ensures(Acc(arg.value_a, 1 / 2) and Acc(arg.value_b, 1 / 3))
76      Ensures(Result() <= arg.value_a + 5 or Result() >= arg.value_b * 5)
77
78      return ClosureCall(f(arg, 5), hof_slot(f)(arg, 5))
79
80
81  def client() -> None:
82
83      arg = Argument(1, 2)
84      assert arg.value_a == 1
85      assert arg.value_b == 2
86
87      f = add
88      y = ClosureCall(f(arg, 3), add)  # type: int
89      assert y == 4
90      assert arg.value_a == 1
91      assert arg.value_b == 2
92
93      f = mul
94      y = ClosureCall(f(arg, 3), mul)
95      assert y == 6
96      assert arg.value_a == 1
97      assert arg.value_b == 2
98
99      if choice():
100         f = add
101     else:
102         f = mul
103
104     @CallSlotProof(add_or_mul(f))
105     def add_or_mul_proof(f: f_type) -> None:
```

```
106
107         @UniversallyQuantified
108         def uq(arg: Argument, x: int) -> None:
109             # Requires(Acc(arg.value_a) and Acc(arg.value_b))
110
111             if f == add:
112                 y = ClosureCall(f(arg, x), add)  # type: int
113             else:
114                 y = ClosureCall(f(arg, x), mul)
115
116             # Ensures(y == arg.value_a + x or y == arg.value_b * x)
117
118     y1 = ClosureCall(f(arg, 3), add_or_mul(f)(arg, 3))  # type: int
119     y2 = ClosureCall(f(arg, 3), add_or_mul(f)(arg, 3))  # type: int
120     assert y1 == 4 or y2 == 6
121     assert y1 == y2
122     assert arg.value_a == 1
123     assert arg.value_b == 2
124     assert y1 == ClosureCall(f(arg, 3), add_or_mul(f)(arg, 3))
125
126     @CallSlotProof(hof_slot(f))
127     def hof_slot_proof(f: f_type) -> None:
128
129         @UniversallyQuantified
130         def uq(arg: Argument, x: int) -> None:
131             # Requires(Acc(arg.value_a) and Acc(arg.value_b))
132
133             if f == add:
134                 y = ClosureCall(f(arg, x), add)  # type: int
135             else:
136                 y = ClosureCall(f(arg, x), mul)
137
138             # Ensures(y <= arg.value_a + x or y >= arg.value_b * x)
139
140     h = hof
141     y = ClosureCall(h(f, arg), hof)
142     assert y <= 6 or y >= 10
143     assert arg.value_a == 1
144     assert arg.value_b == 2
```

LISTING A.8: Pure Example 2

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Automating Modular Reasoning about Higher-Order Functions

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Weber	Benjamin

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Ebikon, 12.11.2017	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.