Modular Specification and Verification of Delegation with SMT Solvers

Modular Specification and Verification of Delegation with SMT Solvers

Ioannis T. Kassios Peter Müller
ETH Zurich

January 3, 2011

Abstract

Delegation is a very common programming idiom, whereby a task is carried out by a statically unknown part of the program. Delegation enhances the modularity and the extensibility of a program, and, for that reason, is the main ingredient of many important design patterns. Unfortunately, delegation complicates specification and verification: the programmer must either rely on unsuitably weak specifications imposed by behavioral subtyping, or compromise automation by resorting to higher-order logic. In this paper, we present an expressive specification and verification methodology, in which partial correctness reasoning about delegation can be carried out in first order logic, using automated SMT solvers.

1 Introduction

Delegation, passing the responsibility of executing a task to statically unknown code, the *delegate*, is a technique used heavily in important design patterns, such as Command, Visitor, Chain of Responsibility, Template [9]. A positive aspect, and a common theme in delegation patterns, is that a delegate can be chosen and/or constructed at run-time, making it possible for a program to exhibit significant extensibility in its behavior.

The downside of delegation is that it creates challenges for modular specification and verification [17, 27], that stem from the fact that the construction and the use of delegates are decoupled. This means that a typical client of a delegate does not have static knowledge of the behavior of the delegate it is using. The question that arises is how to write *modular* specifications that enable the *automatic* verification of the whole program, in the presence of delegation. To demonstrate the problem, we show some examples written in a toy programming language based on *closures*, i.e. first-class procedures.

Example 1 (Commands) *An important delegation-based design pattern is the* Command *pattern. In it, operations that update a document, are stored into*

"command" delegates. The user of the application may generate new commands, e.g. by composing existing ones in various ways.

In our example, a command is a closure that takes a reference to a document, processes it, and returns no result. For simplicity, we use a record that contains a single integer number v to represent the "document".

type *Doc* = { v : **int**; }; //this denotes a mutable record type
type *Command* = *Doc* → (); //this denotes a closure type

The following command divides 100 by the number contained in a document:

val *inverse* = **procedure** (d : *Doc*) { $d.v := 100/d.v$; }

The keyword **val** *introduces a* constant value*, as in Scala.*

For a more involved example, suppose that we want to create a command that adds x to the content of a document, for a given number x. Since x is arbitrary, we need to create a factory *that takes x as a parameter and returns the desired command. We call the factory createAdd:*

val *createAdd* =
procedure (x : **int**) **returns** *Command*
{ **result** := **procedure** (d : *Doc*) { $d.v := d.v + x$; }; }

Here we use the keyword **returns** *in the signature of the procedure to refer to its return type and the keyword* **result** *within its body as the variable that holds its return value.*

Closures capture their lexical environment*. Here, the closure returned by createAdd remembers (*captures*) the value of the formal parameter x even when it goes out of scope.*

The command inverse is easy to decorate with pre- and postconditions. It is not so easy to do the same for the factory procedure createAdd, whose specification must talk about (the specification of) the command that it returns. This is important for the caller, who needs to know when it is permitted to call the command (its precondition), as well as what effects this command has (its postcondition).

This is the first and simplest example of the phenomenon that recurs throughout the paper: the need for specifications that talk about other specifications.

Example 2 (Chain of Responsibility) *We can now proceed to something more complex: we show a factory that takes two different commands and combines them. In particular, our factory takes a boolean predicate on documents, passed as a* pure *(side-effect free) closure b, and two commands $c1$ and $c2$. It produces a command that acts as $c1$ if b is true and as $c2$ otherwise. The name of our procedure is chain:*

```
val chain =
  procedure (b : Doc →f bool, c1 : Command, c2 : Command)
    returns Command
  {
    result := procedure (d : Doc)
              {   if(eval b(d)) { call c1(d); }
                  else { call c2(d); }
              };
  }
```

Here, the notation $\rightarrow_f$ creates types of pure closures. We use the keyword **call** *to invoke non-pure closures and the keyword* **eval** *to evaluate pure closures.*

The factory chain can be used several times to form chains of responsibility*, in which commands may operate only on inputs for which they are responsible, and otherwise delegate the processing to the commands that follow in the chain.*

As a last item in a chain we may use a command that always fails:

```
val fail = procedure (_ : Doc) { abort; }
```

The chain of responsibility factory is even harder to specify, because it is both a factory and a delegant. The same principle however appears again: its specification must talk about the specifications of its inputs and its output.

The classical approach to deal *modularly* with the issue of statically unknown code is *behavioral subtyping* [20]. Behavioral subtyping restricts the specifications of a subclass so that they comply to those of the superclass. A client of an object o, which only knows the dynamic type C of o, can safely assume the specification of C when using o.

Behavioral subtyping alone cannot help us with delegation. Adapted to our examples, behavioral subtyping would entail that all command delegates comply to a common "supertype" specification. But then we would be in a dilemma. If the super type specification is weak enough to allow for all possible command delegates that get created, then it does not suffice to verify the clients. If, on the other hand, we want a specification that is strong enough for the proof of a client, then that may contradict the behavior of some command delegates.

Example 3 (Chain Client) *Consider the following client code, using the definitions of Ex. 1 and 2:*

```
val createChain =
  procedure () returns Command
  {   var sub20, t : Command;
      sub20 := call createAdd(−20);
      t := call chain(function (d1 : Doc) { d1.v > 10 }, sub20, fail);
      result := call chain(function (d1 : Doc) { d1.v ≤ 0 }, inverse, t);
  };
```

```
val client =
  procedure ()
  {   var c : Command; c := call createChain();
      var d : Doc; d := new Doc; d.v := 15;
      call c(d); assert d.v = −5;
      call c(d); assert d.v = −20;
  };
```

where the keyword **function** *creates a pure closure, and the keyword* **var** *a new variable. The reader is invited to ascertain that both assertions in client succeed.*

We want to prove the assertions in client modularly*, i.e., by only looking at its own code and the specifications of the closures it uses. To do that using behavioral subtyping, we must be able to prove that the unknown command object c behaves at least according to the following super type specification:*

```
CommandSuperType(d : Doc)
  requires d.v = 15 ∨ d.v = −5
  ensures (old(d.v = 15) ⇒ d.v = −5) ∧ (old(d.v = −5) ⇒ d.v = −20)
```

However, this specification is already violated by the command inverse, as well as by the middle command-node of the chain c.

On the other hand, if we gave weaker specifications to the command super type, such that all our command delegates comply, then these specifications alone would be insufficient to verify the procedure client.

Current research to attack the problem focuses on higher order logic solutions [11, 12, 2, 13, 30, 22, 16, 29], in which one can quantify over unknown predicates, and therefore over the specifications of delegates. Higher order specifications are expressive enough to solve the problem of specifying delegation-based patterns. However, a significant amount of interaction with a proof checker, such as Coq [10] is required by the user.

1.1 Our Approach

Our focus is different from the approaches presented above: we are interested in *automated verification* using *first-order provers*, in particular, SMT solvers such as Simplify [7] and Z3 [21]. SMT solvers are used in many modern program verifies such as ESC/Java [5], Spec# [1], and VCC [6]. In these systems, all the information that the programmer writes to guide the verifier is part of the program and its specification. The programmer does not interact with the prover in any other way. Supporting this style of automatic verification requires a formalization in first order logic[1].

The idea of our approach is to allow our specifications to be defined in terms of other specifications, by introducing *specification functions*. Specification functions evaluate the pre/postconditions of closures at any state. Using

[1]Integer arithmetic is allowed, since SMT solvers can deal with it effectively.

specification functions, we can write first order logic specifications that are expressive enough to deal with delegation.

Our solution is expressed directly in the Boogie intermediate language [18]. The Boogie verifier translates its programs into SMT-solver input, using a standard wp-calculus.

1.1.1 Contributions of this Paper

To the best of our knowledge, the methodology sketched here is the first *fully formalized* treatment of the problem in first order logic, *targeting SMT solvers*, and with *a full proof of soundness*. The latter is of particular importance, since the idea of specifications that talk about other specifications may introduce inconcistency, an issue that is dealt with in Sect. 2.4.

The concept of specification functions is not new: it appears in [8] and in [24]. However, [8] is not focusing on automated verification and provides a higher order logic formalization, while [24] provides incomplete formalization and no soundness proof.

As a proof-of-concept, the examples presented here, are written directly into the Boogie language and posted on-line at `http://n.ethz.ch/~kassiosi/projects/delegation-bpl-examples/`. The prover behaves as expected with all the examples.

1.1.2 Simplifications made in this Paper

For simplicity, in this paper we introduce no information hiding or data abstraction mechanisms, since we consider this topic orthogonal to the problem at hand. This simplification makes it unnecessary to use special techniques for framing, such as abstract predicates [25], or dynamic frames [14]. In [15], we show how we can add such features to the language. In the paper, we also restrict our interest to partial correctness; reasoning about timing and termination is considered orthogonal.

1.1.3 Summary of the Paper

In Sect. 2, we show the programming and specification language of the paper and we show how to use it to specify delegation based examples. The discussion is informal. The formal semantics of both languages is given in Sect. 3. Sect. 4 shows a translation of the programming/specification language to the intermediate language of the Boogie verifier. In Sect. 5 we prove our verification methodology of Sect. 4 sound with respect to the semantics of Sect. 3.

2 Specification

In this section, we present our programming and specification language. We show how to specify the examples from the Introduction, as well as one extra example. Finally, we also sketch a solution to an important soundness problem.

P	::=	$\overline{TD}\ \overline{D}\ \overline{S}$	program
TD	::=	**type** $I = T$;	type decl.
T	::=	**int** | **bool** | I	type
	|	$(\overline{T}) \to [T\|()]$ | $(\overline{T}) \to_f T$ | { $\overline{I : T;}$ }	
D	::=	**var** $I : T$ | **val** $I = V$	variable/constant
S	::=	$L := E;$ | **if**$(E)\{S\}[$**else**$\{S\}]$	statement
	|	$[L :=]$**call** $E(\overline{E})$; | $L :=$ **new** T;	
	|	**assert** E; | **abort**;	
L	::=	$[E.]I$ | **result**	l-value
E	::=	$L \mid V \mid UO\ E \mid E\ BO\ E \mid$ **eval** $E(\overline{E})$	expression
V	::=	**true** | **false** | N | **null** | PD | FD	value
PD	::=	**procedure** $(\overline{I : T})$ [**returns** T] $\{\overline{D}\ \overline{S}\}$	procedure def.
FD	::=	**function** $(\overline{I : T})$ $\{E\}$	function def.
UO	::=	$\neg \mid + \mid -$	unary operator
BO	::=	$+ \mid - \mid * \mid / \mid \wedge \mid \vee \mid \Rightarrow \mid \Leftarrow$	binary operator

Non-terminal I stands for identifiers. Non-terminal N stands for natural numbers. The operators ::= | [] and the overline are BNF metasymbols. Capital letters are non-terminals. Everything else is literal.

Figure 1: Syntax of the Programming Language

The presentation here is informal. The formal details are provided in Sec. 3.

2.1 Programming Language

In Fig. 1 we see the syntax of our programming language. The informal semantics of most constructs of the language is obvious. Here we explain only the creation and use of *closures*.

Expressions of the programming language can contain *procedure definitions*, introduced with the keyword **procedure**. When a procedure definition gets evaluated, its body is *not executed*. Instead, a new *closure* is created. A closure is a *value* that encapsulates the procedure definition, together with the current *lexical environment*, i.e., all variables and constants that are visible at the point where the procedure definition appears.

It is important to point out the difference between a *procedure definition* and a *closure*. A procedure definition is a *static expression*. A closure is a *value* generated dynamically when a procedure definition is evaluated. One procedure definition may create arbitrarily many closures. For example, consider the procedure definition *inside the body* of *createAdd* in Ex. 1. Each time *createAdd* is invoked, this inner definition creates a *new* closure. A closure gets *invoked* using a **call** command. When the closure is invoked, its body is executed *in the lexical environment that the closure has stored*.

A program may also contain *function definitions*, introduced with the keyword **function**. When a function definition is evaluated, then, like with procedural definitions, the body is *not evaluated*, and instead a *pure closure* is

$$
\begin{array}{lcll}
Q & ::= & [Q.]I \mid \textbf{result} \mid \textbf{true} \mid \textbf{false} & \text{spec. expression} \\
 & \mid & N \mid \textbf{null} \mid \emptyset \mid \{\ \overline{[Q.]I}\ \} \mid \Delta Q & \\
 & \mid & UO\ Q \mid Q\ BO'\ Q \mid \textbf{eval}\ Q(\overline{Q}) & \\
 & \mid & \textbf{old}(Q) \mid \forall I : T \cdot Q \mid \dot{\forall} \cdot Q \mid \textbf{fresh}(Q) & \\
 & \mid & \textbf{pre}(\overline{Q}) \mid \textbf{post}(\overline{Q}) \mid \textbf{mod}(\overline{Q}) \mid \textbf{exec}(\overline{Q}) & \\
BO' & ::= & BO \mid \cup \mid \cap \mid \subseteq & \text{spec. binary operator}
\end{array}
$$

Non-terminal definitions that appear in Fig. 1 are valid here.

Figure 2: Specification Language Syntax

created. The pure closure is a value that encapsulates the function definition together with the current lexical environment. To evaluate the body of a pure closure, we use keyword **eval** within an expression. As with non-pure closures, evaluation is going to happen in the lexical environment that is stored in the pure closure.

For simplicity, *we do not allow* **eval** *expressions or procedure / function definitions within the body of function definitions.* This ensures that *the evaluation of any expression of the programming language terminates and that it creates no new closures.* The principal reason for introducing this restriction, is to avoid solving the problem of circular definitions for both pure and non-pure closures. It is however straightforward to extend the semantics in order to lift this restriction.

2.2 Specification Language

In this section, we equip each procedure definition with classical pre/post-conditions and modifies clauses. We use the standard keywords **requires**, **ensures**, and **modifies** respectively. We also equip functional definitions with preconditions. We introduce and explain our new specification constructs, and we see how they can be used to specify the examples that we have seen so far. The language of specification expressions is shown in Fig. 2.

Specifying procedures with pre/postconditions is standard, and we will not elaborate further on this. In what follows, we show the new constructs of our specification language, that help us with delegation.

2.2.1 The "modifies" Clause.

The treatment of "modifies" clauses is not entirely standard. The **modifies** keyword is followed by a specification expression whose type is *"set of l-values"*. For example, one can write **modifies** $\{x,\ o.y\}$ to allow modification of the value of variable x, as well as field y of record o. As is usual, allocation of new memory is also allowed. We denote the type of "sets of l-values" by **reg** for *region*.

The specification language provides operators to construct expressions of type **reg** and to reason about them. So we are allowed to write "modifies"

clauses such **modifies** $\{x\} \cup \{o.y\}$ or "requires" clauses like **requires** $X \subseteq Y$, where X and Y are specification expressions of type **reg**.

2.2.2 Specification Functions and Related Notation

Specification Functions. To make it possible for specifications of closures to talk about those of other closures, the specification language supports three keywords **pre**, **post**, and **mod**. Their syntax is similar to that of a regular mathematical function. The first parameter is always a closure-valued expression c. Then follows a series of expressions $x_1, x_2, ..$ whose number and types correspond to the parameters expected by the type of c. The operator **post** also takes an expression of the result type of c, if there is one.

The expressions $\mathbf{pre}(c, \overline{x})$ and $\mathbf{post}(c, \overline{x}, r)$ evaluate to a boolean. The former one is true if and only if the precondition of c with parameters the $\overline{x}$ is true in the current state. The latter is true if and only if the precondition of c with parameters the $\overline{x}$ is true *in the current pre-state* and the postcondition of c with parameters the $\overline{x}$ and return value r is true *in the current pre- and post-states*. The expression $\mathbf{mod}(c, \overline{x})$ evaluates to a value of type **reg**. It is equivalent to the "modifies" clause of c with parameters the $\overline{x}$ in the current state. If the precondition is false, then it evaluates to $\emptyset$.

It is important to point out that the expressions $\mathbf{pre}(c, \overline{x})$, $\mathbf{mod}(c, \overline{x})$, and $\mathbf{post}(c, \overline{x}, r)$ *are state-dependent* (in fact, **post** depends on two states), even though the syntax leaves this dependency implicit. Their semantic counterparts are functions whose dependency on the state is explicit. These semantic functions are called *specification functions*.

As an example of the use of specification functions, consider the postcondition of *createAdd* of Ex. 1. It can contain a condition that says that the precondition of the result of *createAdd* is true, if its parameter is non-null:

$$\textbf{ensures } \forall d : \mathit{Doc} \cdot \ d \neq \textbf{null} \ \Rightarrow \ \mathbf{pre}(\textbf{result}, d)$$

Universal State Quantification. The above "ensures" clause is not enough. What it says, is that the return closure can be called *at the current state*, as long as its parameter is non-null. What we really want to say, is that the return closure can be called *at all states*, as long as its parameter is non-null. To introduce universal quantification over all states, we use the symbol $\dot{\forall}$. If Q is a state-dependent expression, then $\dot{\forall} \cdot Q$ is true if and only if Q is true in all states. In our example, we write:

$$\textbf{ensures } \dot{\forall} \cdot \forall d : \mathit{Doc} \cdot \ d \neq \textbf{null} \ \Rightarrow \ \mathbf{pre}(\textbf{result}, d)$$

This says that the precondition of **result** is true in all states, if its parameter is not **null**.

Quantification over two States. In fact, $\dot{\forall}$ is a universal quantification over *two states*, one *prestate* and one *poststate*. Therefore $\dot{\forall}$ introduces a context in which two-state operators, like **post** and **old** can be used. For example, the following postcondition of *createAdd* describes the behavior of the returned closure:

$$\textbf{ensures}\ \dot{\forall} \cdot \forall d : Doc \cdot\ \textbf{post}(\textbf{result}, d)\ \Rightarrow\ d \neq \textbf{null} \wedge d.v = \textbf{old}(d.v) + x$$

This says that if the returned closure is invoked, at *any state*, it will increase the value of $d.v$ by x. Note that we do not need to mention the precondition $d \neq \textbf{null}$ explicitly here. By our convention, for prestates in which $d = \textbf{null}$, the value returned by $\textbf{post}(\textbf{result}, d)$ is **false**, making the implication trivially true.

Fresh closures. When a closure has been created during the execution of an other closure, it is desirable for the client to know so (see Sect. 2.4). To specify this, we use the keyword **fresh**. The keyword **fresh** appears in two-state contexts. It takes one closure-valued argument and asserts that this closure did not exist in the prestate. Closure factories use **fresh** to annotate that their result is fresh; in our example we write:

$$\textbf{ensures fresh}(\textbf{result})$$

Example 4 *Our notation suffices to specify all the examples of Ex. 1 and 2. For example, the procedure definition createAdd of Ex. 1 is annotated as follows:*

```
val createAdd =
  procedure (x : int) returns Command
    ensures ∀̇ · ∀d : Doc · d ≠ null ⇒ pre(result, d)
    ensures ∀̇ · ∀d : Doc · mod(result, d) ⊆ {d.v}
    ensures ∀̇ · ∀d : Doc · post(result, d) ⇒ d.v = old(d.v) + x
    ensures fresh(result)
  { result :=
    procedure (d : Doc)
      requires d ≠ null  modifies {d.v}  ensures d.v = old(d.v) + x
    { d.v := d.v + x; }; }
```

Both procedure definitions are very easily verified in this example. The inner procedure definition is trivial. The outer procedure definition does not *see the code of the inner procedure definition, but still verifies only using the specifications of the inner procedure definition. What is now returned to the client, is a closure together with a description of its behavior.*

The client can take this closure and use it immediately. More interestingly, the closure can be combined into a network of other closures, like the chain of responsibility. To do that, we use procedures such as chain of Ex. 2. Here is how this procedure is annotated with specifications (its body together with the internal specifications is omitted):

```
val chain =
  procedure (b : Doc →f bool, c1 : Command, c2 : Command)
    returns Command
    ensures ∀̇ d : Doc · pre(b, d) ∧ pre(c1, d) ∧ pre(c2, d)
    ensures ∀̇ d : Doc · mod(result, d) ⊆ mod(c1, d) ∪ (mod(c2, d))
    ensures ∀̇ d : Doc·
    ensures fresh(result)
            post(result, d)
      ⇒     (old(eval b(d)) ⇒ post(c1, d)) ∧ (¬old(eval b(d)) ⇒ post(c2, d))
  {...}
```

The specifications of chain and those of individual commands (like inverse from Ex. 1) and command factories (like createAdd from Ex. 1) can be used to create arbitrary complex nets of commands, with known behavior. The procedure createChain of Ex. 3 is an example of such use. The annotation of this command with specification is as follows (the body is again omitted):

```
val createChain =
  procedure () returns Command
    ensures ∀̇ d : Doc · d ≠ null ∧ (d.v < 0 ∨ d.v > 10)  ⇒  pre(result, d)
    ensures ∀̇ d : Doc · mod(result, d) ⊆ {d.v}
    ensures ∀̇ d : Doc·
            post(result, d)
      ⇒         (old(d.v) < 0 ⇒ d.v = 100/old(d.v))
            ∧   (old(d.v) > 10 ⇒ d.v = old(d.v) − 20)
    ensures fresh(result)
  {...}
```

Not only these procedure definitions verify, but they return information strong enough to verify the procedure client of Ex. 3. By contrast, the verification of the following two procedures fails, because they both violate the precondition of our chain.

```
val badClient1 =
  procedure ()
  {   var c : Command; c := call createChain();
      var d : Doc; d := new Doc; d.v := 0;
      call c(d);   //precondition violated
  }
val badClient2 =
  procedure ()
  {   var c : Command; c := call createChain();
      var d : Doc; d := new Doc; d.v := 5;
      call c(d);   //precondition violated
  }
```

The Boogie translation of all the above examples is available at `http://n.ethz.ch/~kassiosi/projects/delegation-bpl-examples/`.

Auxiliary Notation. In what follows, we also make use of two more operators, Δ and **exec**.

The operator Δ takes a one-state **reg**-valued expression Q and produces a two-state specification that guarantees that nothing changes in the heap except the locations described by Q. For example, $\Delta\{x\}$ is a two-state specification that ensures that only the value of variable x may differ between the two states. The operator Δ, like the modifies clause, allows the allocation of new memory.

The keyword **exec** is an abbreviation defined as follows:

$$\mathbf{exec}(c, x_1, .., x_n, r) \quad = \quad (\mathbf{post}(c, x_1, .., x_n, r) \ \wedge \ \Delta(\mathbf{mod}(c, x_1, .., x_n)))$$

In other words, **exec** summarizes the functional and the framing behavior that we observe when we invoke a closure.

2.3 Ghost Parameters

In this section, we implement a custom *while* loop. A formalization of such a general construct would typically need higher-order logic. However, by passing extra information by the client as *ghost parameters*, we can avoid this. In our example, passing the loop invariant suffices to avoid higher-order logic.

Example 5 (Custom While Loop) *The while procedure takes the condition and the loop body as a pure and a non-pure closure respectively. The procedure takes also the loop invariant as a ghost pure closure parameter.*

```
val while =
  procedure (condition : () →f bool , body : () → () , inv : () →f bool)
    requires ∀̇ · pre(inv) ∧ pre(condition) ∧ pre(body)
    requires eval inv()
    requires ∀̇·
            old(eval condition() ∧ eval inv()) ∧ exec(body)
       ⇒    eval (inv) ∧ mod(body) = old(mod(body))
    modifies mod(body)
    ensures ¬eval condition() ∧ eval inv()
  {
    if(eval condition()){ call body(); call while(condition, body, inv); }
  };
```

The specification expresses the usual partial correctness conditions for while loops: the invariant has to hold before the loop and after every iteration. If the loop terminates, its iteration condition will be false and the invariant will hold. Furthermore, the loop promises to respect the frame of its body. For simplicity, we assume that the modifies clause of the body remains constant throughout the iteration, and that all the involved closures have preconditions that are true in all heaps.

The specification of the while procedure allows the clients to reason about invocations of the loop, as in the following example:

```
var i, j, x : int;
i := 0 ; j := 5 ; x := 7;
call while
  (   function (){i < 5} ,
      procedure ()
        modifies {i, j}
        ensures i = old(i) + 1 ∧ j = old(j) − 1
      {i := i + 1; j := j − 1;} ,
      function (){i + j = 5 ∧ i ≤ 5}
  );
assert i = 5 ∧ j = 0 ∧ x = 7;
```

2.4 Avoiding Circular Dependencies between Specification Functions

In our methodology, specifications of procedures talk about specifications of other procedures. If we are not careful, this may cause circularity in the definition of specifications, and therefore inconsistency. The easiest way to produce such an inconsistency is to talk about one's own specification:

```
val f = procedure () requires ¬pre(f) {};
```

It is usually not easy to detect circularity. There might be larger circles than the one we show above, but, more importantly, there may be circles through the heap, as in:

```
var x : () → ();
val f = procedure () requires ¬pre(x) {};
x := f;
```

This problem does not appear in function definitions, because we have explicitly forbidden evaluations within evaluations, ensuring that an **eval** *expression always terminates.*

To prevent the unsoundness, we need to prevent circularity. It suffices to define a well-founded ordering on closures $\sqsubset$ and impose the restriction that the specification functions of a closure c_2 may be depend on the specification functions of another closure c_1, if and only if $c_1 \sqsubset c_2$. The well-foundedness of $\sqsubset$ prevents circular dependency and ensures consistency.

A temptingly easy way to define $c_1 \sqsubset c_2$ is to compare the *size* of the types of c_1 and c_2, as in [15], where the *size* of a type is defined as follows:

$$\begin{array}{l} size(\mathbf{int}) = size(\mathbf{bool}) = size(T) = 1 \text{ for any record type, and pure closure type } T \\ size((T_1, .., T_n) \to R) \;=\; 1 + size(R) + \sum_{i=1}^{n} size(T_i) \\ size((T_1, .., T_n) \to ()) \;=\; 1 + \sum_{i=1}^{n} size(T_i) \end{array}$$

The restriction makes it safe for a specification function to depend on the specification functions of the parameters and the result of the closures it describes.

However, this is too strict in some cases. It is typical in delegation patterns for delegates to depend on other delegates of the same type. Such a situation

exists in Chain of Responsibility. For such cases, we decide to allow c_2 to depend on c_1, if c_1 was created *before* c_2.

In particular, if closure c was the n-th closure to be created during an execution, we say that its *order* is n. We denote the order of a closure c by $order(c)$. We denote the type of a value v by $type(v)$. Finally, we define $\sqsubset$ as follows:

$$\begin{array}{rcll} c_1 \sqsubset c_2 & \Leftrightarrow & & size(type(c_1)) < size(type(c_2)) \\ & & \vee & (size(type(c_1)) = size(type(c_2)) \wedge order(c_1) < order(c_2)) \end{array}$$

This modest improvement is actually quite important. Our ordering allows the specifications of closures to depend on others of the same type, supporting such typical delegation chains, but at the same time gives a simple criterion to ensure absence of circular definitions. All our examples in the previous sections respect the dependency restriction imposed by $\sqsubset$.

The verification system alone can calculate $\sqsubset$ in most cases (see Sect. 4.1), so we felt that it is not necessary to include it to the specification language.

To avoid an easy misunderstanding, we should stress that *we do not impose any restriction on recursion in programs*. We only forbid circularity *in the dependencies of specification functions*. For example, the implementation of *while* in Sect. 2.3, is a recursive procedure.

3 Formal Semantics

Before detailing our automated verification methodology, we present the formal semantics of the specification and programming language of the previous section.

3.1 Notation

The following list summarizes the notation used in this section.

- $[X]Y$ is the set of total maps from X to Y.
- $[X]?Y$ is the set of partial maps from X to Y
- A map m maps x to $m[x]$.
- $\mathsf{Dom}(x)$ is the *domain* of the partial map x.
- $x \mapsto y|z$ is the map such that $m[x] = y$ and $w : \mathsf{Dom}(m) - \{x\} \Rightarrow\ m[w] = z[w]$.
- A tuple is created as follows: $(x; y; ...)$. To get an item from a tuple, we use $\downarrow$
- If X, Y are expressions and x a variable, then $X(Y/x)$ denotes expression X with all free occurrences of x substituted with Y.
- $\mathsf{Id}, \mathsf{Exp}, \mathsf{LV}, \mathsf{Namel}, \mathsf{Stm}, \mathsf{Spec}, \mathsf{Type}$ are the set of identifiers, programming language expressions, l-values, identifier introductions (i.e., **val** and **var** instructions), statements, specification expressions, and programming language types, respectively.
- $\mathbb{N}, \mathbb{Z}, \mathsf{Bool}$ are the sets of naturals, integers, and booleans, respectively.

3.2 State and Value Model

Identifiers. For simplicity, we want to have the property that *every variable and constant declaration in the program introduces a unique identifier*. This is of course not true in our language, because of scoping. However, this technicality can be easily overcome, by translating the program into an equivalent in which all declarations (including formal parameters) introduce *unique* identifiers. From now on, we assume that our programs satisfy this property. The set of all unique identifiers is denoted Id. We make a further distinction into identifiers that denote variables (Id_V) and identifiers that denote constants and formal parameters (Id_C):

$$\mathsf{Id} \;=\; \mathsf{Id}_V \cup \mathsf{Id}_C \qquad \mathsf{Id}_V \cap \mathsf{Id}_C = \emptyset$$

Heaps. Assume an infinite set Loc of *locations* or *addresses*. A *heap* is a partial mapping from locations to *program values*. The set of heaps is denoted Heap and the set of program values by Val:

$$\mathsf{Heap} \;=\; [\mathsf{Loc}]?\mathsf{Val}$$

As we have seen, the lexical environment of a closure gets captured. In practice this entails that some local variables, that would otherwise be on the stack, have to live on the heap. For our purposes, it is enough to assume that there is no stack at all and that *every variable lives on the heap*.

Variable Environments. At any given point of a program, an identifier corresponds to a location. In fact, identifiers are mapped to different locations, according to the closure that is being executed at the time.

This motivates the notion of *variable environment*, a mapping from identifiers to locations. The set of variable environments is denoted Env. An important property of a variable environment is that no two distinct identifiers are mapped to the same location (never two variables refer to the same memory location):

$$\mathsf{Env} \;\subseteq\; \{\; e : [\mathsf{Id}_V]\mathsf{Loc} \mid \forall i, j : \mathsf{Dom}(e) \cdot \;\; i \neq j \;\;\Rightarrow\;\; e[i] \neq e[j] \;\}$$

Notice that we are using $\subseteq$ instead of equality here. The reason is that *records* are also mappings from identifiers to locations, and we want to distinguish records from variable environments.

Also notice that we are formalizing environments as *total* mappings, in contrast to heaps, which are formalized as partial mappings. Total mappings are formally simpler and therefore preferable, whenever possible. We are never interested in the *domain* of an environment (this part is taken care of by the type checker before us), so we can use total mappings here. In the case of heaps, this is not so: any allocation of new memory must ensure that the new locations are not already used in the heap for something else.

Values. A *value* is a boolean, an integer, **null**, a record, or a closure. Notice that, like environments, records are simply total mappings from identifiers to distinct locations. The set of all closures is denoted Cl:

$$\begin{array}{l} \mathsf{Rec} \subseteq [\mathsf{Id}_V]\mathsf{Loc} \\ \mathsf{Val} = \mathbb{Z} \cup \mathsf{Bool} \cup \{\mathbf{null}\} \cup \mathsf{Rec} \cup \mathsf{Cl} \end{array}$$

Furthermore, we want the following properties: (a) no two distinct records may use the same location:

$$\forall r, r' : \mathsf{Rec},\ i : \mathsf{Dom}(r),\ j : \mathsf{Dom}(r') \cdot\ r \neq r' \lor i \neq j\ \Rightarrow\ r[i] \neq r[j]$$

and (b) no record uses a location used by a variable environment:

$$\forall r : \mathsf{Rec},\ e : \mathsf{Env}, i : \mathsf{Dom}(r),\ j : \mathsf{Dom}(e) \cdot\ r[i] \neq e[j]$$

Notice that the first axiom guarantees also that one record maps distinct identifiers to distinct locations, a property that we also met in environments.

Constant Environments. Our language supports constants. These include the formal parameters of procedure and function definitions. We store these in a seperate map, called the *constant environment*. The type of this map is denoted Const. A constant environment maps identifiers to values.

$$\mathsf{Const} = [\mathsf{Id}_C]\mathsf{Val}$$

During execution, we have a pair of a variable environment E and a constant environment C. We refer to the pair $(\mathsf{E}; \mathsf{C})$ as the *environment* of the execution.

Closures. A first-order formalization does not permit storing the actual body of a closure as a value. Instead, each closure contains a token that uniquely represents the *procedure or function definition* from which it was created. We call this token the *key* of the procedure or function definition. Keys are used to connect closures to their body and their specifications. We denote the set of keys CK.

Apart from the key, a closure carries its *lexical environment*. This means that a closure has a variable environment component that remembers all the variables that the closure captures, as well as a constant environment component that remembers all the constants that it captures. Non-pure closures also carry their order, for reasons that we saw in Sect. 2.4. The definition of Cl is thus:

$$\begin{array}{l} \mathsf{Cl} = \mathsf{NPCl} \cup \mathsf{PCl} \\ \mathsf{NPCl} = \mathsf{CK} \times \mathsf{Env} \times \mathsf{Const} \times \mathbb{N} \\ \mathsf{PCl} = \mathsf{CK} \times \mathsf{Env} \times \mathsf{Const} \end{array}$$

We define

$$\mathit{key}(c) = c\downarrow_1 \quad \mathit{env}(c) = c\downarrow_2 \quad \mathit{cst}(c) = c\downarrow_3 \quad \mathit{order}(c) = c\downarrow_4$$

For any closure type T, we assume the existence of a unique closure key $uninit_T$, that describes the value of an uninitialized T-typed variable. The precondition of any closure coming from $uninit_T$ is **false** (i.e., no uninitialized closure may be invoked).

States. The state of execution is described by the *current heap*, the *current lexical environment* (i.e. a variable and a constant environment), the *current return value* (corresponding to the value of the special variable **result**), and a number which is greater than the order of each closure ever created, called the *max-order* of the state. The set of states is denoted by Σ:

$$\Sigma \;=\; \mathsf{Heap} \times \mathsf{Env} \times \mathsf{Const} \times \mathsf{Val} \times \mathbb{N}$$

3.3 Semantics of Programs

Expressions. The denotation of expressions is a function $[\![]\!]_{\mathcal{E}}$ that takes an expression and a state. The function produces a value. Expressions have also a subtle side effect: they create and insert new closures in the heap. This means that they change the max-order of the state. The function $[\![]\!]_{\mathcal{M}}$ is a function that takes an expression and a state and returns the new max-order after the evaluation of the expression. The function is $[\![]\!]_{\mathcal{L}}$ takes an l-value and a state and produces the location pointed to by the l-value:

$$\begin{array}{l} [\![\cdot]\!]_{\mathcal{E}} \;:\; \mathsf{Exp} \times \Sigma \to \mathsf{Val} \\ [\![\cdot]\!]_{\mathcal{M}} \;:\; \mathsf{Exp} \times \Sigma \to \mathbb{N} \\ [\![\cdot]\!]_{\mathcal{L}} \;:\; \mathsf{LV} \times \Sigma \to \mathsf{Loc} \end{array}$$

The semantics of expressions can be seen in Fig. 3. Obvious cases, like unary and binary operators, are omitted.

Statements. For the semantics of statements, we define a *uninitialized value* $def(T)$ for each type T, as follows:

$def(T) = \mathbf{null}$	if T is a record type
$def(T) = (uninit_T; \emptyset; \emptyset; 0)$	if T is a non-pure closure type
$def(T) = (uninit_T; \emptyset; \emptyset)$	if T is a pure closure type
$def(T) : T$	otherwise (underspecified)

In Fig. 4 we show a big-step operational semantics of our language. The described ternary relation $\rightsquigarrow$ takes a pre-state, an identifier introduction or statement, and a post-state:

$$(\cdot, \cdot) \rightsquigarrow \cdot \;:\; \Sigma \times (\mathsf{Namel} \cup \mathsf{Stm}) \times \Sigma$$

Again, trivial cases, like conditionals and sequential composition, are omitted.

It is instructive to see the semantics of the closure invocation. The actual parameters are evaluated in the calling environment E, C. Their values are accumulated to the constant environment c of the closure that is being invoked, so

L-values, constants and **result**. For simplicity, we assume that no procedure/function definitions appear in l-values.

$$I : \mathsf{Id}_V \Rightarrow [\![I]\!]_{\mathcal{L}}((\mathsf{H};\mathsf{E};\mathsf{C};\mathsf{R};\mathsf{M})) = \mathsf{E}[I] \qquad [\![E.I]\!]_{\mathcal{L}}(\sigma) = [\![E]\!]_{\mathcal{E}}(\sigma)[I]$$
$$[\![L]\!]_{\mathcal{E}}(\sigma) = \mathsf{H}[[\![L]\!]_{\mathcal{L}}(\sigma)] \qquad [\![L]\!]_{\mathcal{M}}((\mathsf{H};\mathsf{E};\mathsf{C};\mathsf{R};\mathsf{M})) = \mathsf{M}$$
$$I : \mathsf{Id}_C \Rightarrow [\![I]\!]_{\mathcal{E}}((\mathsf{H};\mathsf{E};\mathsf{C};\mathsf{R};\mathsf{M})) = \mathsf{C}[I] \qquad I : \mathsf{Id}_C \Rightarrow [\![I]\!]_{\mathcal{M}}((\mathsf{H};\mathsf{E};\mathsf{C};\mathsf{R};\mathsf{M})) = \mathsf{M}$$
$$[\![\mathbf{result}]\!]_{\mathcal{E}}((\mathsf{H};\mathsf{E};\mathsf{C};\mathsf{R};\mathsf{M})) = \mathsf{R} \qquad [\![\mathbf{result}]\!]_{\mathcal{M}}((\mathsf{H};\mathsf{E};\mathsf{C};\mathsf{R};\mathsf{M})) = \mathsf{M}$$

Closures: Assume that

- The function definition **function** $(\overline{x})\{B\}$ has unique closure key u.
- The procedure definition **procedure** $(\overline{y})\{S\}$ has unique closure key u'.

Then:

$$\begin{array}{ll}
[\![\mathbf{function}\ (\overline{x})\{B\}]\!]_{\mathcal{E}}((\mathsf{H};\mathsf{E};\mathsf{C};\mathsf{R};\mathsf{M})) & = (u;\mathsf{E};\mathsf{C}) \\
[\![\mathbf{function}\ (\overline{x})\{B\}]\!]_{\mathcal{M}}((\mathsf{H};\mathsf{E};\mathsf{C};\mathsf{R};\mathsf{M})) & = \mathsf{M} \\
[\![\mathbf{procedure}\ (\overline{y})\{S\}]\!]_{\mathcal{E}}((\mathsf{H};\mathsf{E};\mathsf{C};\mathsf{R};\mathsf{M})) & = (u';\mathsf{E};\mathsf{C};\mathsf{M}) \\
[\![\mathbf{procedure}\ (\overline{y})\{S\}]\!]_{\mathcal{M}}((\mathsf{H};\mathsf{E};\mathsf{C};\mathsf{R};\mathsf{M})) & = \mathsf{M}+1
\end{array}$$

Pure Closure Evaluation: Assume that

- The function definition **function** $(x_1,..,x_n)\{B\}$ has unique closure key u.
- $[\![E]\!]_{\mathcal{E}}((\mathsf{H};\mathsf{E};\mathsf{C};\mathsf{R};\mathsf{M})) = (u;e;c)$ and $[\![E]\!]_{\mathcal{M}}((\mathsf{H};\mathsf{E};\mathsf{C};\mathsf{R};\mathsf{M})) = m_1$.
- For all i: $[\![E_i]\!]_{\mathcal{E}}((\mathsf{H};\mathsf{E};\mathsf{C};\mathsf{R};m_i)) = v_i$ and $[\![E_i]\!]_{\mathcal{M}}((\mathsf{H};\mathsf{E};\mathsf{C};\mathsf{R};m_i)) = m_{i+1}$.

Then:

$$\begin{array}{ll}
[\![\mathbf{eval}\ E(E_1,..,E_n)]\!]_{\mathcal{E}}((\mathsf{H};\mathsf{E};\mathsf{C};\mathsf{R};\mathsf{M})) & = [\![B]\!]_{\mathcal{E}}((\mathsf{H};e;x_1 \mapsto v_1|..|x_n \mapsto v_n|c;\mathsf{R};m_{n+1})) \\
[\![\mathbf{eval}\ E(E_1,..,E_n)]\!]_{\mathcal{M}}((\mathsf{H};\mathsf{E};\mathsf{C};\mathsf{R};\mathsf{M})) & = m_{n+1}
\end{array}$$

Figure 3: Semantics of Expressions

The semantics of statements and name introductions are as follows:

$$\frac{I : \mathsf{Id}_V \quad l : \mathsf{Loc} - \mathsf{Dom}(\mathsf{oH}) \quad \mathsf{E} = I \mapsto l | \mathsf{oE} \quad \mathsf{H} = l \mapsto \mathit{def}(T) | \mathsf{oH}}{(\mathbf{var}\ I : T\ , (\mathsf{oH}; \mathsf{oE}; \mathsf{C}; \mathsf{R}; \mathsf{M})) \rightsquigarrow (\mathsf{H}; \mathsf{E}; \mathsf{C}; \mathsf{R}; \mathsf{M})}$$

$$\frac{I : \mathsf{Id}_C \quad \mathsf{C} = I \mapsto [\![V]\!]_{\mathcal{E}}((\mathsf{H}; \mathsf{E}; \mathsf{oC}; \mathsf{R}; \mathsf{oM})) | \mathsf{oC} \quad \mathsf{M} = [\![V]\!]_{\mathcal{M}}((\mathsf{H}; \mathsf{E}; \mathsf{oC}; \mathsf{R}; \mathsf{oM}))}{(\mathbf{val}\ I = V\ , (\mathsf{H}; \mathsf{E}; \mathsf{oC}; \mathsf{R}; \mathsf{oM})) \rightsquigarrow (\mathsf{H}; \mathsf{E}; \mathsf{C}; \mathsf{R}; \mathsf{M})}$$

$$\frac{\begin{array}{c} l = [\![L]\!]_{\mathcal{L}}((\mathsf{oH}; \mathsf{E}; \mathsf{C}; \mathsf{R}; \mathsf{oM})) : \mathsf{Dom}(\mathsf{H}) \quad \mathsf{H} = l \mapsto [\![E]\!]_{\mathcal{E}}((\mathsf{oH}; \mathsf{E}; \mathsf{C}; \mathsf{R}; \mathsf{oM})) | \mathsf{oH} \\ \mathsf{M} = [\![E]\!]_{\mathcal{M}}((\mathsf{oH}; \mathsf{E}; \mathsf{C}; \mathsf{R}; \mathsf{oM})) \end{array}}{(L := E\ , (\mathsf{oH}; \mathsf{E}; \mathsf{C}; \mathsf{R}; \mathsf{oM})) \rightsquigarrow (\mathsf{H}; \mathsf{E}; \mathsf{C}; \mathsf{R}; \mathsf{M})}$$

$$\frac{\mathsf{R} = [\![E]\!]_{\mathcal{E}}((\mathsf{H}; \mathsf{E}; \mathsf{C}; \mathsf{oR}; \mathsf{oM})) \quad \mathsf{M} = [\![E]\!]_{\mathcal{M}}((\mathsf{H}; \mathsf{E}; \mathsf{C}; \mathsf{oR}; \mathsf{oM}))}{(\mathbf{result} := E\ , (\mathsf{H}; \mathsf{E}; \mathsf{C}; \mathsf{oR}; \mathsf{oM})) \rightsquigarrow (\mathsf{H}; \mathsf{E}; \mathsf{C}; \mathsf{R}; \mathsf{M})}$$

For the invocation statement, we show only the case with a return value. Assume that the procedure definition **procedure** $(x_1, .., x_n)$ **returns** T $\{S\}$ has unique closure key u. Assume also that the return variable for the procedure definition is rv.

$$\frac{\begin{array}{c} [\![E]\!]_{\mathcal{E}}((\mathsf{oH}; \mathsf{E}; \mathsf{C}; \mathsf{R}; \mathsf{oM})) = (u; e; c; o) \quad c_1 = c \quad m_1 = [\![E]\!]_{\mathcal{M}}((\mathsf{oH}; \mathsf{E}; \mathsf{C}; \mathsf{R}; \mathsf{oM})) \\ \mathit{rcl} = [\![L]\!]_{\mathcal{L}}((\mathsf{oH}; \mathsf{E}; \mathsf{C}; \mathsf{R}; \mathsf{oM})) : \mathsf{Dom}(\mathsf{oH}) \\ \forall i \cdot c_{i+1} = x_i \mapsto [\![E_i]\!]_{\mathcal{E}}((\mathsf{oH}; \mathsf{E}; \mathsf{C}; \mathsf{R}; m_i)) | c_i \\ \forall i \cdot m_{i+1} = [\![E_i]\!]_{\mathcal{M}}((\mathsf{oH}; \mathsf{E}; \mathsf{C}; \mathsf{R}; m_i)) \\ (S,\ (\mathsf{oH}; e; c_{n+1}; \mathit{dummy}; m_{n+1})) \rightsquigarrow (h'; e'; c'; r; \mathsf{M}) \\ \mathsf{H} = \mathit{rcl} \mapsto r | h' \end{array}}{(L := \mathbf{call}\ E(E_1, .., E_n)\ , (\mathsf{oH}; \mathsf{E}; \mathsf{C}; \mathsf{R}; \mathsf{oM})) \rightsquigarrow (\mathsf{H}; \mathsf{E}; \mathsf{C}; \mathsf{R}; \mathsf{M})}$$

Figure 4: Semantics of Statements

that the body of the closure can see them. The body of the closure S is executed in the environment $e; c_{n+1}$, where c_{n+1} is the constant environment of the closure *together with all the actual parameters as constants*. After the execution, the caller resumes in its own environment $\mathsf{E}; \mathsf{C}$, which has not changed.

3.4 Semantics of Specifications

The denotation of specification expressions is given by a function $[\![]\!]_{\mathcal{S}}$. This takes a specification expression and translates it against two states[2].

Notice that we only need one variable environment and one constant environment (the environment changes during the execution of the specified closure are ignored in the caller). We also require only one return value.

Furthermore, to implement the restrictions of Sect. 2.4, we introduce one extra argument, the *current closure*. This is the closure whose specification is currently being evaluated. In Sect. 4.2, we see how this parameter is being used to avoid circular definitions.

A specification expression returns either a value (for pre/postconditions, this value is a boolean), or a set of locations (for example, the semantics of a modifies clause is a set of locations).

[2]For one-state specifications, the second state is omitted.

The discussion above explains the, rather complicated, signature of $[\![]\!]_{\mathcal{S}}$:

$$[\![\cdot]\!]_{\mathcal{S}} \ : \ \mathsf{Spec} \times \mathsf{Heap} \times \mathsf{Heap} \times \mathbb{N} \times \mathbb{N} \times \mathsf{Env} \times \mathsf{Const} \times \mathsf{Val} \times \mathsf{NPCI} \to \mathsf{Val} \cup 2^{\mathsf{Loc}}$$

For each non-pure closure type T, we introduce three *specification functions* pre_T, $post_T$ and mod_T. For example, if $T = (T_1, .., T_n) \to R$, then

$$\begin{array}{l} pre_T \ : \ \mathsf{Heap} \times \mathbb{N} \times \mathsf{NPCI} \times T_1 \times .. \times T_n \to \mathsf{Bool} \\ post_T \ : \ \mathsf{Heap} \times \mathsf{Heap} \times \mathbb{N} \times \mathbb{N} \times \mathsf{NPCI} \times T_1 \times .. \times T_n \times R \to \mathsf{Bool} \\ mod_T \ : \ \mathsf{Heap} \times \mathbb{N} \times \mathsf{NPCI} \times T_1 \times .. \times T_n \to 2^{\mathsf{Loc}} \end{array}$$

The function pre_T and mod_T take one heap and one max-order as parameter, because they are to be evaluated in one state. The function $post_T$ takes two such pairs.

The function pre_T is defined accordingly for pure closure types T. Together we define a function $eval_T$:

$$\begin{array}{l} pre_T \ : \ \mathsf{Heap} \times \mathbb{N} \times \mathsf{PCI} \times T_1 \times .. \times T_n \to \mathsf{Bool} \\ eval_T \ : \ \mathsf{Heap} \times \mathbb{N} \times \mathsf{PCI} \times T_1 \times .. \times T_n \to R \end{array}$$

We show how specification functions are axiomatized in Sect. 4.2. For the time being, we use them as uninterpreted symbols in our semantics.

The semantics of specification expressions is given in Fig. 5 (everything except specification functions) and Fig. 6 (specification functions). Constants, unary, and binary operators are omitted.

4 Verification

In our verification technique, we are using Boogie, a tool which is based on SMT solvers to verify programs against their specification.

Our technique makes several passes through the program to be verified:

- The first pass is trivial: it changes the identifiers of the program so that each declaration creates a unique identifier, and it assigns a unique key to each procedure and function definition.
- The second pass discovers *environment invariants* for each scope that occurs in the program.
- The third pass creates an *axiomatization* that links the specifications that appear in any procedure / function definition to the corresponding specification functions.
- The fourth pass creates a Boogie procedure *per each procedure definition* in the program and annotates it with appropriate specifications. This step is not needed for function definitions (the axiomatization of *eval* in the third step is enough).

Identifiers and Record Fields (a helper function $[\![\cdot]\!]_{\mathcal{LS}}$ plays the same role as $[\![\cdot]\!]_{\mathcal{L}}$ for programming language expressions):

$$
\begin{array}{l}
I : \mathsf{Id}_V \;\Rightarrow\; [\![I]\!]_{\mathcal{LS}}(\mathsf{H},\mathsf{oH},\mathsf{M},\mathsf{oM},\mathsf{E},\mathsf{C},\mathsf{R},D) \;=\; \mathsf{E}[I] \\
[\![Q.I]\!]_{\mathcal{LS}}(\mathsf{H},\mathsf{oH},\mathsf{M},\mathsf{oM},\mathsf{E},\mathsf{C},\mathsf{R},D) \;=\; [\![Q]\!]_{\mathcal{LS}}(\mathsf{H},\mathsf{oH},\mathsf{M},\mathsf{oM},\mathsf{E},\mathsf{C},\mathsf{R},D)[I] \\
[\![L]\!]_{\mathcal{S}}(\mathsf{H},\mathsf{oH},\mathsf{M},\mathsf{oM},\mathsf{E},\mathsf{C},\mathsf{R},D) \;=\; \mathsf{H}[[\![L]\!]_{\mathcal{LS}}(\mathsf{H},\mathsf{oH},\mathsf{M},\mathsf{oM},\mathsf{E},\mathsf{C},\mathsf{R},D)] \\
[\![\{Q_1,..,Q_n\}]\!]_{\mathcal{S}}(\mathsf{H},\mathsf{oH},\mathsf{M},\mathsf{oM},\mathsf{E},\mathsf{C},\mathsf{R},D) \;= \\
\quad \{\; [\![Q_i]\!]_{\mathcal{LS}}(\mathsf{H},\mathsf{oH},\mathsf{M},\mathsf{oM},\mathsf{E},\mathsf{C},\mathsf{R},D) \mid i : \{1,..,n\} \;\} \\
I : \mathsf{Id}_C \;\Rightarrow\; [\![I]\!]_{\mathcal{S}}(\mathsf{H},\mathsf{oH},\mathsf{M},\mathsf{oM},\mathsf{E},\mathsf{C},\mathsf{R},D) \;=\; \mathsf{C}[I] \\
[\![\mathbf{result}]\!]_{\mathcal{S}}(\mathsf{H},\mathsf{oH},\mathsf{M},\mathsf{oM},\mathsf{E},\mathsf{C},\mathsf{R},D) \;=\; \mathsf{R}
\end{array}
$$

Special Specification Constructs:

$$
\begin{array}{l}
[\![\Delta Q]\!]_{\mathcal{S}}(\mathsf{H},\mathsf{oH},\mathsf{M},\mathsf{oM},\mathsf{E},\mathsf{C},\mathsf{R},D) \;= \\
\quad \forall l : \mathsf{Dom}(\mathsf{oH}) - [\![Q]\!]_{\mathcal{S}}(\mathsf{oH},\mathsf{oH},\mathsf{oM},\mathsf{oM},\mathsf{E},\mathsf{C},\mathsf{R},D) \cdot \mathsf{H}[l] = \mathsf{oH}[l] \\
[\![\mathbf{old}(Q)]\!]_{\mathcal{S}}(\mathsf{H},\mathsf{oH},\mathsf{M},\mathsf{oM},\mathsf{E},\mathsf{C},\mathsf{R},D) \;=\; [\![Q]\!]_{\mathcal{S}}(\mathsf{oH},\mathsf{oH},\mathsf{oM},\mathsf{oM},\mathsf{E},\mathsf{C},\mathsf{R},D) \\
[\![\dot{\forall} \cdot Q]\!]_{\mathcal{S}}(\mathsf{H},\mathsf{oH},\mathsf{M},\mathsf{oM},\mathsf{E},\mathsf{C},\mathsf{R},D) \;= \\
\quad \forall h', h : \mathsf{Heap}, m', m : \mathbb{N} \cdot [\![Q]\!]_{\mathcal{S}}(h', h, m', m, \mathsf{E},\mathsf{C},\mathsf{R},D) \\
[\![\mathbf{fresh}(Q)]\!]_{\mathcal{S}}(\mathsf{H},\mathsf{oH},\mathsf{M},\mathsf{oM},\mathsf{E},\mathsf{C},\mathsf{R},D) \;= \\
\quad (\mathsf{oM} \leq \mathit{order}([\![Q]\!]_{\mathcal{S}}(\mathsf{H},\mathsf{oH},\mathsf{M},\mathsf{oM},\mathsf{E},\mathsf{C},\mathsf{R},D)) < \mathsf{M})
\end{array}
$$

Universal Quantification over Values:

$$
\begin{array}{l}
[\![\forall x : T \cdot Q]\!]_{\mathcal{S}}(\mathsf{H},\mathsf{oH},\mathsf{M},\mathsf{oM},\mathsf{E},\mathsf{C},\mathsf{R},D) \;= \\
\quad \forall v : T \cdot [\![Q(v/x)]\!]_{\mathcal{S}}(\mathsf{H},\mathsf{oH},\mathsf{M},\mathsf{oM},\mathsf{E},\mathsf{C},\mathsf{R},D)
\end{array}
$$

Figure 5: Semantics of Specifications without Specification Functions

Finally, the usual Boogie process is followed: using a standard wp-calculus, all Boogie procedures are translated into SMT formulas, to be fed into an SMT solver, which returns the answer.

Our technique concerns the first four passes. The first one is trivial. In this section, we detail the next three passes. The soundness of verification is proved in Sec. 5.

4.1 Environment Assumptions

The purpose of this pass is to address an incompleteness that stems from the fact that Boogie supports only flat procedures, while our source language supports nesting. Consider the following code:

var x : **int**;
val g = **procedure** (z : **int**) { $x := x + z$ };
val f = **procedure** () { **val** h = **procedure** (z : **int**) { $g(z)$; $x := x + 1$; }; };

The code is legal. In particular, the procedure h accesses legaly the visible variable x and the visible constant g. However, x and g are not declared as part of the signature of h. The Boogie procedure that corresponds to h must get all information about x and g by somewhere else.

Assume that u is the key of a procedure definition. In this section, we define a predicate ψ_u that collects this static information about the environment of

Pure Closures. Assume that

- $[\![Q]\!]_{\mathcal{S}}(\mathsf{H}, \mathsf{oH}, \mathsf{M}, \mathsf{oM}, \mathsf{E}, \mathsf{C}, \mathsf{R}, D) = (u; e; c)$
- $\mathit{type}((u; e; c)) = T = (T_1, .., T_n) \rightarrow_f RT$
- For all i, $[\![Q_i]\!]_{\mathcal{S}}(\mathsf{H}, \mathsf{oH}, \mathsf{M}, \mathsf{oM}, \mathsf{E}, \mathsf{C}, \mathsf{R}, D) = v_i$

. Then:

$$\begin{array}{rcl} [\![\mathbf{eval}\ Q(Q_1, .., Q_n)]\!]_{\mathcal{S}}(\mathsf{H}, \mathsf{oH}, \mathsf{M}, \mathsf{oM}, \mathsf{E}, \mathsf{C}, \mathsf{R}, D) & = & \mathit{eval}_T(\mathsf{H}, \mathsf{M}, (u; e; c), \overline{v}) \\ [\![\mathbf{pre}(Q, Q_1, .., Q_n)]\!]_{\mathcal{S}}(\mathsf{H}, \mathsf{oH}, \mathsf{M}, \mathsf{oM}, \mathsf{E}, \mathsf{C}, \mathsf{R}, D) & = & \mathit{pre}_T(\mathsf{H}, \mathsf{M}, (u; e; c), \overline{v}) \end{array}$$

Non-Pure Closures. Assume that

- $[\![Q]\!]_{\mathcal{S}}(\mathsf{H}, \mathsf{oH}, \mathsf{M}, \mathsf{oM}, \mathsf{E}, \mathsf{C}, \mathsf{R}, D) = D' = (u; e; c; m)$
- $\mathit{type}(D') = T = (T_1, .., T_n) \rightarrow RT$
- For all i, $[\![Q_i]\!]_{\mathcal{S}}(\mathsf{H}, \mathsf{oH}, \mathsf{M}, \mathsf{oM}, \mathsf{E}, \mathsf{C}, \mathsf{R}, D) = v_i$

Then:

$$[\![\mathbf{pre}(Q, Q_1, .., Q_n)]\!]_{\mathcal{S}}(\mathsf{H}, \mathsf{oH}, \mathsf{M}, \mathsf{oM}, \mathsf{E}, \mathsf{C}, \mathsf{R}, D) = \begin{cases} \mathit{pre}_T(\mathsf{H}, \mathsf{M}, D', \overline{v}) & \text{if } D' \sqsubset D \\ \mathbf{false} & \text{o.w.} \end{cases}$$

$$[\![\mathbf{mod}(Q, Q_1, .., Q_n)]\!]_{\mathcal{S}}(\mathsf{H}, \mathsf{oH}, \mathsf{M}, \mathsf{oM}, \mathsf{E}, \mathsf{C}, \mathsf{R}, D) = \begin{cases} \mathit{mod}_T(\mathsf{H}, \mathsf{M}, D', \overline{v}) & \text{if } D' \sqsubset D \\ \emptyset & \text{o.w.} \end{cases}$$

$$[\![\mathbf{post}(Q, Q_1, .., Q_n, R)]\!]_{\mathcal{S}}(\mathsf{H}, \mathsf{oH}, \mathsf{M}, \mathsf{oM}, \mathsf{E}, \mathsf{C}, \mathsf{R}, D) = \begin{cases} \mathit{post}_T(\mathsf{H}, \mathsf{oH}, \mathsf{M}, \mathsf{oM}, D', \overline{v}, \mathsf{R}) & \text{if } D' \sqsubset D \\ \mathbf{false} & \text{o.w.} \end{cases}$$

The case for closures without return values just omits the return value from *post*. The operator **exec** is defined by:

$$\mathbf{exec}(Q, Q_1, .., Q_n, R) \equiv \mathbf{post}(Q, Q_1, .., Q_n, R) \wedge \Delta(\mathbf{mod}(Q, Q_1, .., Q_n))$$

Figure 6: Semantics of Specifications - Specification Functions

the definition. We then include this information as a precondition of the Boogie procedure that corresponds to the definition.

The information that we gather is:

- Which variables are visible.
- The values of constants.
- The relation between the *orders* of constant non-pure closure constants.

The signature of ψ_u is:

$$\psi_u : \mathsf{Heap} \times \mathsf{Env} \times \mathsf{Const} \to \mathsf{Bool}$$

and we define it as a conjunct of various conditions. In particular, $\psi_u(\mathsf{H}, \mathsf{E}, \mathsf{C})$ contains:

- $\mathsf{E}[x] : \mathsf{Dom}(\mathsf{H})$, for any visible variable.
- $\mathsf{C}[x] = v$, for any visible constant declaration **val** $x = v$; where v is a primitive value.
- $\mathit{key}(\mathsf{C}[x]) = u'$, for any visible constant declaration **val** $x = d$; where d is a procedure or function definition with key u'.
- Let c_1, c_2 be two non-pure-closure-valued visible constants. If c_1 is introduced before c_2 in the same scope, or if c_1 is introduced in a scope that includes the scope of c_2, then $\psi_u(\mathsf{H}, \mathsf{E}, \mathsf{C})$ contains the conjunct $\mathit{order}(\mathsf{C}[c_1]) < \mathit{order}(\mathsf{C}[c_2])$.
- Let f be a non-pure-closure-valued formal parameter of a scope that includes u. Let c be a non-pure-closure-valued constant introduced in a scope in which f is visible. Then $\psi_u(\mathsf{H}, \mathsf{E}, \mathsf{C})$ contains the conjunct $\mathit{order}(\mathsf{C}[f]) < \mathit{order}(\mathsf{C}[c])$.

The intention is that at the beginning of the execution of a closure with key u, the conjunct $\psi_u(\mathsf{H}, \mathsf{E}, \mathsf{C})$ is true (where $\mathsf{H}, \mathsf{E}, \mathsf{C}$ constitute the heap and the environment of the current state). This is proved in Thm. 1 below.

Going back to our example, assume that g has key u_g and f has key u_f. Then $\psi_u(\mathsf{H}, \mathsf{E}, \mathsf{C})$ is given by:

$$\begin{array}{rcll} \psi_u(\mathsf{H}, \mathsf{E}, \mathsf{C}) & = & (& \mathsf{E}[x] : \mathsf{Dom}(\mathsf{H}) \ \wedge \ \mathit{key}(\mathsf{C}[g]) = u_g \ \wedge \ \mathit{key}(\mathsf{C}[f]) = u_f \\ & & \wedge & \mathit{key}(\mathsf{C}[h]) = u \ \wedge \ \mathit{order}(\mathsf{C}[h]) > \mathit{order}(\mathsf{C}[f]) > \mathit{order}(\mathsf{C}[g]) \ ) \end{array}$$

4.2 Axiomatization of Specification Functions

In this pass, we axiomatize the specification functions. This means, we provide Boogie axioms that associate the specification functions pre_T, post_T, and mod_T with the respective specifications declared in the procedure and function definitions. This axiomatization is used in the semantics of Fig. 6.

For example, consider the following procedure definition f, with key u (in context **var** x : **int**;):

procedure $(y : \mathbf{int})$ **modifies** $\{x\}$ **ensures** $x = \mathbf{old}(x) + y$ { $x := x + y$; }

This procedure definition triggers the following axiomatization:

$$\forall h : \mathsf{Heap},\ d : \mathsf{NPCl},\ m : \mathbb{N},\ y : \mathbb{Z} \cdot key(d) = u \Rightarrow pre_{\mathbf{int}\to()}(h, m, d, y)$$
$$\forall h : \mathsf{Heap},\ d : \mathsf{NPCl},\ m : \mathbb{N},\ y : \mathbb{Z} \cdot key(d) = u \Rightarrow mod_{\mathbf{int}\to()}(h, m, d, y) \subseteq \{env(d)[x]\}$$
$$\forall h, oh : \mathsf{Heap},\ m, om : \mathbb{N},\ d : \mathsf{NPCl},\ y : \mathbb{Z} \cdot$$
$$key(d) = u \Rightarrow\ (post_{\mathbf{int}\to()}(h, oh, m, om, d, y)\ =\ (h[env(d)[x]] = oh[env(d)[x]] + y))$$

that is, for any closure d generated by this procedure definition:

- the precondition of d is always true.
- the modifies clause of d is always $\{x\}$.
- the postcondition of d is equivalent to "the new value of x is equal to the old value of x plus y".

4.2.1 Non-Pure Closures

Let u be a key for a procedure definition with type[3] $T = (T_1, .., T_n) \to R$. Let P, Q, M be respectively the pre/postcondition and the modifies clause of that procedure definition. Let $x_1, .., x_n$ be the formal parameters and r be the return variable of the procedure definition. Then the specification function axiomatization generates the following axioms:

$$\forall h : \mathsf{Heap},\ m : \mathbb{N},\ d : \mathsf{NPCl},\ \overline{v : T} \cdot key(d) = u\ \Rightarrow$$
$$pre_T(h, m, d, \overline{v})\ =\ [\![P]\!]_{\mathcal{S}}(h, h, m, m, env(d),\ x_1 \mapsto v_1|..|x_n \mapsto v_n|cst(d)\ , dummy, d)$$

$$\forall h : \mathsf{Heap},\ m : \mathbb{N},\ d : \mathsf{NPCl},\ \overline{v : T} \cdot key(d) = u\ \wedge\ pre_T(h, m, d, \overline{v})\ \Rightarrow$$
$$mod_T(h, m, d, \overline{v})\ =\ [\![M]\!]_{\mathcal{S}}(h, h, m, m, env(d),\ x_1 \mapsto v_1|..|x_n \mapsto v_n|cst(d)\ , dummy, d)$$

$$\forall h : \mathsf{Heap},\ m : \mathbb{N},\ d : \mathsf{NPCl},\ \overline{v : T} \cdot key(d) = u\ \wedge\ \neg pre_T(h, m, d, \overline{v})\ \Rightarrow$$
$$mod_T(h, m, d, \overline{v})\ =\ \emptyset$$

$$\forall h, oh : \mathsf{Heap},\ m, om : \mathbb{N},\ d : \mathsf{NPCl},\ \overline{v : T},\ r : R \cdot key(d) = u\ \Rightarrow$$
$$post_T(h, oh, m, om, d, \overline{v}, r)$$
$$=\quad pre_T(oh, om, d, \overline{v}) \wedge [\![Q]\!]_{\mathcal{S}}(h, oh, m, om, env(d),\ x_1 \mapsto v_1|..|x_n \mapsto v_n|cst(d)\ , r, d)$$

Each one of these axioms defines one specification function in terms of the actual specification written in the code. The value *dummy* is never used. The axioms respect our convention that **post** returns **false** and that **mod** returns $\emptyset$, when the precondition is not satisfied.

The axiomatization presented here completes the formalization of the solution which was sketched in Sect. 2.4. The trick lies in the final parameter of the semantics, the "current closure". Suppose that we evaluate a specification annotation of a closure d_1. Each time a **pre**, **post**, or **mod** subexpression is evaluated, the semantics of Fig. 6 equate it to the corresponding specification function *pre*, *post*, or *mod*, only if their closure-parameter d_2 is "lower" than the current one in the $\sqsubset$-ordering, i.e. $d_2 \sqsubset d_1$. In turn, the axiomatization presented here, changes the "current closure" argument to d_2, when evaluating the related specification annotation of d_2. Since $\sqsubset$ is well-founded, this can happen only finitely many times, which avoids definitional circles.

[3] For procedures without return value, the simplifications are immediate.

4.2.2 Pure Closures

Let u be a key for a function definition with type $T = (T_1, .., T_n) \rightarrow_f R$. Let P, B be resp. the precondition and the body of that function definition. Let $x_1, .., x_n$ be the formal parameters of the function definition. Then the specification function axiomatization generates the following axioms:

$$\begin{array}{l} \forall h : \mathsf{Heap},\ m : \mathbb{N},\ d : \mathsf{PCl},\ \overline{v : T} \cdot key(d) = u\ \Rightarrow \\ \quad pre_T(h, m, d, \overline{v}) \\ = \quad [\![P]\!]_{\mathcal{S}}(h, h, m, m, env(d),\ x_1 \mapsto v_1 | .. | x_n \mapsto v_n | cst(d)\ , dummy_1, dummy_2) \end{array}$$

$$\begin{array}{l} \forall h : \mathsf{Heap},\ m : \mathbb{N},\ d : \mathsf{PCl},\ \overline{v : T} \cdot key(d) = u\ \Rightarrow \\ \quad eval_T(h, m, d, \overline{v}) \\ = \quad [\![B]\!]_{\mathcal{S}}(h, h, m, m, env(d),\ x_1 \mapsto v_1 | .. | x_n \mapsto v_n | cst(d)\ , dummy_1, dummy_2) \end{array}$$

The values $dummy_1$ and $dummy_2$ are never used.

4.3 Translation to Boogie Procedures

The final pass of the process, translates every procedure definition into a Boogie procedure. It is this Boogie procedure that is used by the verifier to check the correctness of procedure definitions. The verification process is done independently of the other procedure definitions. Function definitions do not need this pass: their axiomatization as Boogie functions is enough.

The Boogie procedure operates on four global parameters $\mathsf{H}, \mathsf{E}, \mathsf{C}, \mathsf{M}$, which constitute the state of the program (except for the return value). It takes a closure-valued parameter *me*, which represents the current closure. If the procedure definition has a return value, then the Boogie procedure has a return value called *result*.

Let u be a key for a procedure definition with return type R. Let P, Q, M be the pre-/post-condition and the modifies clause of that procedure definition resp. Let S be the body of the procedure definition. The translation of the procedure definition in Boogie is:

$$\begin{array}{l} \textbf{procedure}\ proc\_u(me : \mathsf{NPCl})\ \textbf{returns}\ (\textbf{result} : R) \\ \quad \textbf{requires}\ key(me) = u\ \wedge\ 0 \leq order(me) < \mathsf{M}\ \wedge\ \psi_u(\mathsf{H}, \mathsf{E}, \mathsf{C}); \\ \quad \textbf{requires}\ [\![P]\!]_{\mathcal{S}}(\mathsf{H}, \mathsf{H}, \mathsf{M}, \mathsf{M}, \mathsf{E}, \mathsf{C}, dummy, me); \\ \quad \textbf{modifies}\ \mathsf{H}, \mathsf{E}, \mathsf{C}, \mathsf{M}; \\ \quad \textbf{ensures}\ [\![Q]\!]_{\mathcal{S}}(\mathsf{H}, \textbf{old}(\mathsf{H}), \mathsf{M}, \textbf{old}(\mathsf{M}), \mathsf{E}, \mathsf{C}, \textbf{result}, me); \\ \quad \textbf{ensures} \\ \quad\quad \forall l : \mathsf{Dom}(\textbf{old}(\mathsf{H})) - [\![M]\!]_{\mathcal{S}}(\textbf{old}(\mathsf{H}), \textbf{old}(\mathsf{H}), \textbf{old}(\mathsf{M}), \textbf{old}(\mathsf{M}), \mathsf{E}, \mathsf{C}, dummy, me)) \cdot \\ \quad\quad\quad \mathsf{H}[l] = \mathsf{oH}[l]; \\ \{ \\ \quad Translate(S) \\ \} \end{array}$$

where *Translate* takes the body S written in the source language and translates it into Boogie.

The first Boogie precondition is a conjunctions of facts that are already known about the closure under specification: its key, the fact that its order is below the maximum order of the state, and the environment assumptions $\psi_u(\mathsf{H}, \mathsf{E}, \mathsf{C})$, that we constructed in the second pass of the translation phase (Sect. 4.1). The formal parameters of the procedure definition are in C. The second Boogie precondition is the

translation of the precondition of the procedure definition as written in the source language.

The Boogie modifies clause does *not* correspond to the source language modifies clause. It only asserts that the state is being changed by this Boogie procedure, but provides no other detail. The translation of the source language modifies clause comes later in the Boogie postconditions.

The first Boogie postcondition is the translation of the postcondition of the procedure definition as written in the source language. The second Boogie postcondition is the translation of the source language modifies clause: it expresses that no allocated location changes its value, unless it is included in the source language modifies clause, evaluated at the prestate.

The body of the Boogie procedure is a translation of the source language statements of the procedure definition, into their Boogie counterparts. This translation follows exactly the operational semantics of Fig. 4, except for non-pure closure invocation and pure closure evaluation. Therefore, in our discussion of *Translate*, we omit all other statements.

Translation of Invocations and Evaluations. To translate closure invocations and evaluations *modularly*, we assume no knowledge of the respective procedure or function definition bodies. Therefore, we cannot use the semantics of Fig. 3 and Fig. 4. We use specification functions instead. In particular, the source code $L := \textbf{call}\ E(E_1, .., E_n)$; gets translated into the Boogie code shown in Fig. 7, while the translation of $\textbf{eval}\ E(E_1, .., E_n)$ is shown in Fig. 8. Thm. 3 below, shows that this translation is correct.

Let us focus on the translation of closure invocation (similar comments apply to closure evaluation). The steps taken are no different than the normal Boogie invocation. First, the parameters are calculated. Then the precondition is asserted and the state is havoced. Finally, the postcondition and the frame condition are assumed and the returned value is written to the proper place in the state. The difference here is that the pre/postconditions and the modifies clause are statically unknown, which is the reason why we use the mechanism of specification functions to refer to them dynamically.

The assumption of the history invariants at the end of the invocation is easily justified by the semantics of the language.

5 Soundness

In this section, we prove the soundness of our methodology in a series of theorems. By the term *soundness* we mean two things:

- The translation process cannot cause inconsistencies to the background axiomatization. An inconsistency to the axioms used by the verifier, means that all proof obligations are trivially true, i.e., all programs verify trivially (this is also called a *miracle*).
- The translation process is faithful to the semantics of the programming and the specification language. In particular, in our partial correctness methodology, if a program verifies, this means that executing any closure from a state that satisfies its precondition, then the execution will either fail to terminate, or terminate in a state that satisfies the postcondition of the closure.

// retrieve the closure
var $c := [\![E]\!]_{\mathcal{E}}((\mathsf{H};\mathsf{E};\mathsf{C};\mathbf{result};\mathsf{M}))$;

// evaluate the parameters
var $v_1 := [\![E_1]\!]_{\mathcal{E}}((\mathsf{H};\mathsf{E};\mathsf{C};\mathbf{result};\mathsf{M}));\ \mathsf{M} := [\![E_1]\!]_{\mathcal{M}}((\mathsf{H};\mathsf{E};\mathsf{C};\mathbf{result};\mathsf{M}))$;
..
var $v_n := [\![E_n]\!]_{\mathcal{E}}((\mathsf{H};\mathsf{E};\mathsf{C};\mathbf{result};\mathsf{M}));\ \mathsf{M} := [\![E_n]\!]_{\mathcal{M}}((\mathsf{H};\mathsf{E};\mathsf{C};\mathbf{result};\mathsf{M}))$;

// assert precondition and save the pre-state
assert $pre_T(\mathsf{H},\mathsf{M},c,v_1,..,v_n)$;
var $\mathsf{oH} := \mathsf{H}$; **var** $\mathsf{oM} := \mathsf{M}$;

// havoc the state
havoc H,M;

// assume the postcondition and the frame condition
var $rv : R$;
assume $post_T(\mathsf{H},\mathsf{oH},\mathsf{M},\mathsf{oM},c,v_1,..,v_n,rv)$;
assume $\forall l : \mathsf{Dom}(\mathsf{oH}) - mod_T(\mathsf{oH},\mathsf{oM},c,v_1,..,v_n) \cdot \mathsf{H}[l] = \mathsf{oH}[l]$;

// assume history invariants
assume $\mathsf{Dom}(\mathsf{H}) \supseteq \mathsf{Dom}(\mathsf{oH})$;
assume $\mathsf{M} \geq \mathsf{oM}$;

// return value
var $rl := [\![L]\!]_{\mathcal{L}}((\mathsf{oH};\mathsf{E};\mathsf{C};\mathbf{result};\mathsf{oM}))$;
$\mathsf{H}[rl] := rv$;

where T is the type of the closure and R is the return type.

Figure 7: Translation of invocation $L :=$ **call** $E(E_1,..,E_n)$; to Boogie

Before evaluating an expression that contains **eval** $E(E_1,..,E_n)$, we check if the precondition of that evaluation is true:

var $c := [\![E]\!]_{\mathcal{E}}((\mathsf{H};\mathsf{E};\mathsf{C};\mathbf{result};\mathsf{M}))$;
var $v_1 := [\![E_1]\!]_{\mathcal{E}}((\mathsf{H};\mathsf{E};\mathsf{C};\mathbf{result};\mathsf{M}));\ \mathsf{M} := [\![E_1]\!]_{\mathcal{M}}((\mathsf{H};\mathsf{E};\mathsf{C};\mathbf{result};\mathsf{M}))$;
..
var $v_n := [\![E_n]\!]_{\mathcal{E}}((\mathsf{H};\mathsf{E};\mathsf{C};\mathbf{result};\mathsf{M}));\ \mathsf{M} := [\![E_n]\!]_{\mathcal{M}}((\mathsf{H};\mathsf{E};\mathsf{C};\mathbf{result};\mathsf{M}))$;
assume $pre_T(\mathsf{H},\mathsf{M},c,v_1,..,v_n)$;

The translation of **eval** $E(E_1,..,E_n)$ within the expression is simply:

$eval_T(\mathsf{H},\mathsf{M},c,v_1,..,v_n)$

where T is the type of the closure.

Figure 8: Translation of evaluation to Boogie

5.1 Free Conditions

A *free condition* is an *assumption* that we insert at a given point in a program, to facilitate the verification. This assumption may be used freely by the prover, as a lemma. A free condition must be justified by the semantics of the programming language: we must be able to prove, using the semantics of the programming language, that the condition is always true at the point where it is inserted. Otherwise, we risk miraculous behavior.

In our translation process, we insert the following free conditions:

- In Boogie procedures we introduce three free preconditions: the first one specifies the key for the "me" closure, the second one specifies that its order does not exceed M, and the third one is the environment assumption $\psi_u(\mathsf{H}, \mathsf{E}, \mathsf{C})$.
- After a closure invocation, we assume two history invariants: first that the domain of H has not decreased, and second that M has not decreased.

Except for the environment assumption, it is easy to see that the rest of our free conditions hold at the points of their insertion. The following theorem proves the correctness of environment assumptions.

Theorem 1 *Let u be a closure key and c a closure that was created from u. Let H be the heap and E, C the environment at the beginning of the execution (or evaluation) of u. Then $\psi_u(\mathsf{H}, \mathsf{E}, \mathsf{C})$.*

Proof: We only prove the second conjunct, i.e., $\mathsf{C}[x] = v$ for a visible definition **val** $x = v$; The proofs of the other conjuncts are similar.

We first make one remark. Assume the execution of **val** $x = v$;. The semantics of Fig. 4, guarantee that $\mathsf{C}[x] = v$ at any subsequent point in the scope of the definition of x.

Let D be a procedure or function definition, from which x is visible. Let k be the nesting scope of D, and k' the nesting scope of **val** $x = v$; Since x is visible from D, there is some procedure definition D' at nesting scope k', that contains D, and that appears after **val** $x = v$;. In the trivial case, $k = k'$, and D coincides with D'.

Notice that, to reach D', the introduction **val** $x = v$; must be executed first. By the above remark, $\mathsf{C}[x] = v$ when we reach D'.

We now prove the theorem by induction on $k' - k$.

Base Case: In the base case, D and D' coincide. This proves that $\mathsf{C}[x]$ when D is reached. Let d be a closure created by D. Then, at the point of the creation of d, $cst(d) = \mathsf{C}$, which means that an execution of d starts at a state where $\mathsf{C}[x] = v$.

Induction Hypothesis: If $k - k' = n$, then any closure created from D, starts executing at a state where $\mathsf{C}[x] = v$.

Induction Step: We prove that $\mathsf{C}[x] = v$ at D, if $k - k' = n + 1$. Let D'' be the immediate procedure definition containing D. To create any closure d from D, one needs to execute a closure d'' created by D''. By the induction hypothesis, any such d'' starts at a state where $\mathsf{C}[x] = v$. When D is reached, the semantics of Fig. 4 guarantee that $\mathsf{C}[x] = v$ is true. Let d be a closure created by D. Then, at the point of the creation of d, $cst(d) = \mathsf{C}$, which means that an execution of d starts at a state where $\mathsf{C}[x] = v$. □

5.2 No Miracles

To prove that our source language cannot produce any miraculous behavior, we must prove our axiomatization correct. Our axiomatization is non-trivial, since it involves mutually recursive definitions (Sect. 4.2.1). We prove here that these definitions are consistent, which prevents miracles.

Theorem 2 *The axiomatization of specification functions presented in Sect. 4.2.1 is consistent.*

Proof: The axiomatization is a mutually recursive definition. To prove it consistent, one must find a well-founded ordering and a measure in which the ordering decreases. We choose $\sqsubset$ as our well-founded ordering. It is enough to prove that the specification functions of any closure d depend only on specification functions of closures d' such that $d' \sqsubset d$.

Let d be a closure. Its specification functions depend on a semantic function f whose "current closure" is d. By the semantics of Fig. 5, we see that f does not depend on the specification functions of any closure d' such that $d' \not\sqsubset d$. In fact, the only interesting cases are the semantics of the keywords **pre**, **mod**, and **post**, which depend on the specification functions of a closure d' only if $d' \sqsubset d$. □

As we have already commented, the specification functions of pure closures are not recursive, and therefore trivially consistent.

5.3 Translation Correctness

Finally, we prove our translation correctness theorem, as explained in the introduction of this section. The theorem uses the weakest precondition semantics of Boogie. The "weakest precondition" function *wp*, takes a series of Boogie statements $\mathcal{S}$ and a predicate on two states Q and returns a predicate on one state $P = wp(\mathcal{S}, Q)$. The condition $P(\sigma)$ is necessary to ensure that the execution of $\mathcal{S}$ from state σ to state σ' satisfies $Q(\sigma', \sigma)$.

Theorem 3 *Assume*

- $\mathcal{P}$ *is a piece of arbitrary type-correct code in the source language.*
- $\mathcal{M}$ *is the translation of* $\mathcal{P}$ *into Boogie.*
- $\mathcal{M}$ *verifies.*
- σ *is a state and* $(\mathcal{P}, \sigma) \rightsquigarrow \sigma'$.
- P *is a predicate on one state and* Q *is a predicate on two states.*
- $\forall \sigma : \Sigma \cdot P(\sigma) \Rightarrow wp(\mathcal{M}, Q)(\sigma)$.
- $P(\sigma)$.

Then $Q(\sigma', \sigma)$.

Proof: This theorem can be proved by induction on the derivation trees of the operational semantics. As we have said, the Boogie translation follows the operational semantics, except in closure invocation and evaluation. Therefore, it suffices to prove this theorem for the closure invocation (using the inductive assumption that the theorem holds for any code higher in the derivation tree). We only give the proof for

invocation of non-pure closures here. For simplicity, we omit the passing of formal parameters and the return variable (we only consider the type $() \to ()$). We also omit the proof for the evaluation of pure closures, which is similar but simpler.

Our statement is **call** E; Let $d = [\![E]\!]_{\mathcal{E}}(\sigma)$ be the closure under invocation. Let its closure key be u, its environment e_d, c_d its body S, its precondition PRE, its modifies clause MOD, and its postcondition $POST$.

The translation is:

var $d := [\![E]\!]_{\mathcal{E}}(\sigma)$;
assert $pre_{()\to()}(\mathsf{H}, \mathsf{M}, d)$;
var $\mathsf{oH} := \mathsf{H}$; **var** $\mathsf{oM} := \mathsf{M}$;
havoc H, M;
assume $post_{()\to()}(\mathsf{H}, \mathsf{oH}, \mathsf{M}, \mathsf{oM}, d)$;
assume $\forall l : \mathsf{Dom}(\mathsf{oH}) - mod_{()\to()}(\mathsf{oH}, \mathsf{oM}, d) \cdot \mathsf{H}[l] = \mathsf{oH}[l]$;
assume $\mathsf{Dom}(\mathsf{H}) \supseteq \mathsf{Dom}(\mathsf{oH})$;
assume $\mathsf{M} \geq \mathsf{oM}$;

From the fact that $P(\sigma)$ we derive that

$$wp(\mathcal{M}, Q)(\sigma)$$

which is equivalent to

$$\begin{array}{lll} & pre_{()\to()}(\mathsf{oH}, \mathsf{oM}, d) & \\ \wedge & \forall h, e, c, r, m \cdot & \\ & \quad post_{()\to()}(h, \mathsf{oH}, m, \mathsf{oM}, d) & \\ & \wedge \quad (\forall l : \mathsf{Dom}(\mathsf{oH}) - mod_{()\to()}(\mathsf{oH}, \mathsf{oM}, d) \cdot h[l] = \mathsf{oH}[l]) & \\ & \wedge \quad e = \mathsf{oE} \ \wedge \ c = \mathsf{oC} \ \wedge \ r = \mathsf{oR} \ \wedge \ \mathsf{Dom}(h) \supseteq \mathsf{Dom}(\mathsf{oH}) \ \wedge \ m \geq \mathsf{oM} & \\ & \Rightarrow \quad Q((h; e; c; r; m), (\mathsf{oH}; \mathsf{oE}; \mathsf{oC}; \mathsf{oR}; \mathsf{oM})) & \end{array}$$

where $d = [\![E]\!]_{\mathcal{E}}(\sigma)$ and $\sigma = (\mathsf{oH}; \mathsf{oC}; \mathsf{oE}; \mathsf{oR}; \mathsf{oM})$.
The above formula is equivalent to

$$\begin{array}{lll} & pre_{()\to()}(\mathsf{oH}, \mathsf{oM}, d) & \\ \wedge & \forall h, m \cdot & \\ & \quad post_{()\to()}(h, \mathsf{oH}, m, \mathsf{oM}, d) & \\ & \wedge \quad (\forall l : \mathsf{Dom}_{()\to()}(\mathsf{oH}) - mod(\mathsf{oH}, \mathsf{oM}, d) \cdot h = \mathsf{oH}[l]) & \\ & \wedge \quad \mathsf{Dom}(h) \supseteq \mathsf{Dom}(\mathsf{oH}) \ \wedge \ m \geq \mathsf{oM} & \\ & \Rightarrow \quad Q((h; \mathsf{oE}; \mathsf{oC}; \mathsf{oR}; m), (\mathsf{oH}; \mathsf{oE}; \mathsf{oC}; \mathsf{oR}; \mathsf{oM})) & \end{array}$$

Call the above condition A.

By the axiomatization of Sect. 4.2.1, the first conjunct of A gives us

$$[\![PRE]\!]_{\mathcal{S}}(\mathsf{oH}, \mathsf{oH}, \mathsf{oM}, \mathsf{oM}, e_d, c_d, dummy, d)$$

Notice that the precondition of the Boogie translation of u is satisfied, (as discussed in Sect. 5.1, the free preconditions are given by the semantics of the programming language). Therefore, we conclude (inductively) that the postcondition is also satisfied. In particular, we have

$$[\![POST \ \wedge \ \Delta(\mathbf{old}(MOD))]\!]_{\mathcal{S}}(\mathsf{H}, \mathsf{oH}, \mathsf{M}, \mathsf{oM}, e_d, c_d, dummy, d)$$

where $\sigma' = (\mathsf{H}, \mathsf{E}, \mathsf{C}, \mathsf{R}, \mathsf{M})$.

By the axiomatization of Sect. 4.2.1, and since $pre_{()\to()}(\mathsf{oH}, \mathsf{oM}, d)$, we get:

$$post_{()\to()}(\mathsf{H}, \mathsf{oH}, \mathsf{M}, \mathsf{oM}, d) \ \wedge \ (\forall l : \mathsf{Dom}(\mathsf{oH}) - mod_{()\to()}(\mathsf{oH}, \mathsf{oM}, d) \cdot \mathsf{H} = \mathsf{oH}[l])$$

We now instantiate A for $h = \mathsf{H}$ and $m = \mathsf{M}$. The conditions

$$\mathsf{Dom}(\mathsf{H}) \supseteq \mathsf{Dom}(\mathsf{oH}) \ \wedge \ \mathsf{M} \geq \mathsf{oM}$$

are free (see Sect. 5.1). This implies

$$[\![Q]\!]_{\mathcal{S}}((\mathsf{H}; \mathsf{oE}; \mathsf{oC}; \mathsf{oR}; \mathsf{M}), (\mathsf{oH}; \mathsf{oE}; \mathsf{oC}; \mathsf{oR}; \mathsf{oM}))$$

By the semantics of the invocation, $\mathsf{oE} = \mathsf{E}$, $\mathsf{oC} = \mathsf{C}$, and $\mathsf{oR} = \mathsf{R}$, which gives

$$[\![Q]\!]_{\mathcal{S}}((\mathsf{H}; \mathsf{E}; \mathsf{C}; \mathsf{R}; \mathsf{M}), (\mathsf{oH}; \mathsf{oE}; \mathsf{oC}; \mathsf{oR}; \mathsf{oM}))$$

which is what we wanted to prove. □

6 Experience

We translated all the examples shown in this paper into Boogie and posted them online. The translation process was straightforward. All the examples behaved as expected.

The ease by which Boogie verifies the examples is due to the fact that our methodology is very SMT-friendly. Not only do we formalize everything in first order logic, but we also avoid existential quantification, a feature with which SMT solvers cannot cope easily. A frequent complaint of users of SMT solvers is the difficulty of finding the correct *triggers*[4] [7]. We found that this is not a problem in our methodology, due to the fact that our axioms are *definitional*, which means that they have obvious triggers. For example, a definitional axiom $\forall x, y, z \cdot f(x, y, z) = ...$ has $f(x, y, z)$ as an obvious triggering expression.

Most of the time, a delegate is either executed immediately, passed over, or inserted into a larger network of delegates. These are uses that our methodology handles very well, as we also demonstrate in our on-line Boogie code. Other design patterns are also very naturally expressed using our methodology are *Builder* (the *createChain* procedure of Ex. 3 is actually a builder object), *Factory* (*createAdd*, *chain*, and *createChain* are factories), and *Visitor*.

However, there are delegation-based design patterns, such as the *Template*, for which our current technique is usually weak. The problem with such patterns is that intermediate states of the execution of a delegate often become important, something that our formalism does not handle well. *Model programs* [27] handle this situation better. Inspired by model programs, we are currently extending our work to that direction. A general exposition of this extension is beyond the scope of the present

[4] Triggers are auxiliary annotation that helps the prover instantiate the correct universal quantifications.

paper, but an interesting example, that even goes beyond the expressive power of model programs, appears in Sect. 2.3, where the power of delegation is used to implement a custom "while" loop.

7 Related Work

In this section, we discuss other verification techniques that deal with delegation.

7.1 Techniques based on Higher-Order Logic

The most comprehensive framework for dealing with imperative higher-order language features is that of Yoshida, Honda and Berger [11, 12, 2, 13, 30], a total correctness Hoare-logic based framework. While it covers all technical aspects of delegates, the language provides no explicit abstract state or other means of human annotation, relying instead on existential quantifications. We are not aware of any use of the framework in a verification tool.

Delegation is also treated in *Hoare Type Theory* [22], a framework that unifies types with Hoare-logic specifications and targets a higher-order imperative language. Hoare Type Theory is supported by the tool Ynot [23] which delegates proof obligations to the Coq theorem prover. The programmer interacts with Coq to construct proofs of correctness. Instead, in our approach, all the interaction with the prover happens only with annotations within the programming language.

Recently, Higher-Order Separation Logic [3] has been employed to deal with delegates. An exploration of higher-order frame rules appears in [4], but the language does not involve first-class delegates and the modularity is static. A stronger language is used in [16], where several design patterns are elegantly formalized. In [29] a more comprehensive framework is presented for dealing with delegates in a Higher-Order Separation Logic context, since it addresses issues like state capturing, state abstraction, and framing. We are not aware of a tool that supports [29].

The idea of *specification functions* appears in an early theoretical work [8], which however does not escape higher-order logic.

7.2 Approaches not using Higher-Order Logic

A very important work for delegation-based patterns is JML's *model programs* [27]. Model programs are specifications that are written in programming style. The problem with this approach is that it sometimes generates too strong specifications. In particular, a procedure that implements a model program style of specification follows necessarily the syntactic structure of the model program. This may rule out many interesting implementations, but, more importantly, is based on the fact that the control flow of the procedure is statically known. In the Chain of Resposibility pattern of Ex. 2, we have shown a case where the control flow of a delegate is dynamically created, which rules out the model-program specification style. The same is true for the "while loop" of Ex. 5. It must be admitted however that model programs tend to be concise and easy to use, and therefore a better match for patterns, in which the control flow is known statically and is fairly complex (the Template pattern usually generates such cases). In our future work, we wish to extend our methodology to support the model program-style of specification together with our specification functions.

Our technique of reducing delegates to first order objects is a variant of Reynold's *defunctionalization* [26]. Reynold's work provides the mechanism by which the transformation from higher-order programs to first order programs can be realized, but does not concern itself with specification and verification. It seems that defunctionalization is the primary tool that takes us from the higher-order logic of [8] to first-order logic.

The mechanism of specification functions appears also in [24]. However our work goes further in providing a complete formalization and a soundness proof. This is especially important for the methodology, because of the unsoundness issue discussed in Sect. 2.4.

8 Conclusion

We have introduced the first fully formalized modular specification and verification methodology for delegation, that is friendly to automated SMT-based program verifiers. We have proved the soundness of the methodology, and we have used it successfully to some non trivial examples, which are posted on-line at `http://n.ethz.ch/~kassiosi/projects/delegation-bpl-examples/`.

We plan to apply the ideas from this paper to the verification of Scala's traits, whose "super-calls" resemble delegation. Another direction for future work is the combination of our technique with verification methodologies based on permissions such as implicit dynamic frames [28, 19], as well as with the approach of model programs [27]. An integration of our specification and verification technique to a full-blown object-oriented language like C# or Scala is also future work.

References

[1] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# specification language: an overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *CASSIS'04*, volume 3362 of *Lecture Notes In Computer Science*, pages 49–69. Springer-Verlag, 2004.

[2] M. Berger, K. Honda, and N. Yoshida. A logical analysis of aliasing for higher-order imperative functions. In *ICFP'05*, pages 280–293, 2005.

[3] B. Biering, L. Birkedal, and N. Torp-Smith. BI-hyperdoctrines, higher-order separation logic, and abstraction. *ACM Transactions on Programming Languages and Systems*, 29(5), 2007.

[4] L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation logic typing and higher order frame rules for Algol-like languages. *Logical Methods in Computer Science*, 2(5:1), 2006.

[5] P. Chalin, J.R. Kiniry, G. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *FMCO'05*, volume 4111 of *Lecture Notes In Computer Science*, pages 342–363. Springer-Verlag, 2005.

[6] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskał, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *Lecture Notes In Computer Science*, 2009.

[7] Detlefs D, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005.

[8] G. W. Ernst, J. K. Navlakha, and W. F. Ogden. Verification of programs with procedure-type parameters. *Acta Informatica*, 18(2), 1982.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.

[10] N. A. Hamid. Theorem proving with the COQ proof assistant: Tutorial presentation. *Journal of Computing Sciences in Colleges*, 24(2), 2008.

[11] K. Honda. From process logic to program logic. In *ICFP'04*, pages 163–17. ACM, 2004.

[12] K. Honda and N. Yoshida. A compositional logic for polymorphic higher-order functions. In *PPDP'04*, pages 191–202. ACM, 2004.

[13] K. Honda, N. Yoshida, and M. Berger. An observationally complete program logic for imperative higher-order functions. In *LICS'05*, pages 270–279, 2005.

[14] I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM'06*, volume 4085 of *Lecture Notes In Computer Science*, pages 268–283. Springer-Verlag, 2006.

[15] I. T. Kassios and P. Müller. Specification and verification of closures. Technical Report 660, Dept. of Computer Science, ETH Zurich, 2010.

[16] N. Krishnaswami, J. Aldrich, L. Birkedal, K. Svendsen, and A. Buisse. Design patterns in separation logic. In *TLDI '09*, pages 105–116. ACM, 2009.

[17] G. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object oriented programs. *Formal Aspects of Computing*, 19(2):159–189, 2007.

[18] K. R. M. Leino. This is Boogie 2. Working Draft - available at `http://-research.microsoft.com/en-us/um/people/leino/papers.html`, 2008.

[19] K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In G. Castagna, editor, *ESOP'09*, volume 5502 of *Lecture Notes In Computer Science*, pages 378–393. Springer-Verlag, 2009.

[20] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.

[21] L. Moura and N. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS'08*, volume 4963 of *Lecture Notes In Computer Science*, pages 337–340. Springer-Verlag, 2008.

[22] A. Nanevski, G. Morrisett, and L. Birkedal. Hoare Type Theory, polymorphism and separation. *Journal of Functional Programming*, 18(5-6):865–911, 2008.

[23] A. Nanevski, G. Morrisett, L. Birkedal, A. Shinnar, Paul Govereau, and L. Birkedal. Ynot: Reasoning with the awkward squad. In *ICFP '08*, 2008.

[24] M. Nordio, C. Calcagno, B. Meyer, P. Müller, and J. Tschannen. Reasoning about function objects. In J. Vitek, editor, *TOOLS'10*, volume 6141 of *Lecture Notes In Computer Science*. Springer-Verlag, 2010.

[25] M. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL'05*, pages 247–258, 2005.

[26] J. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM '72*, pages 717–740. ACM, 1972.

[27] S. M. Shaner, G. Leavens, and D. Naumann. Modular verification of higher order methods with mandatory calls specified by model programs. In *OOPSLA'07*, pages 351–368. ACM, 2007.

[28] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In S. Drossopoulou, editor, *ECOOP'09*, volume 5653 of *Lecture Notes In Computer Science*, pages 148–172. Springer-Verlag, 2009.

[29] K. Svedsen, L. Birkedal, and M. Parkinson. Verifying generics and delegates. In *ECOOP'10*, volume 6183 of *Lecture Notes In Computer Science*. Springer-Verlag, 2010.

[30] N. Yoshida, K. Honda, and M. Berger. Logical reasoning for higher-order functions with local state. *CoRR*, abs/0806.2448, 2008.