Specification and Verification of Closures

Ioannis T. Kassios and Peter Müller

ETH Zurich, Switzerland
{ioannis.kassios , peter.mueller} @inf.ethz.ch

Abstract. Closures, first-class citizen procedures that are able to capture their lexical environment, increase the expressiveness of object-oriented languages such as C#, Scala, and various dynamic languages. However, closures make program specification and verification more difficult. For instance, a verification methodology must allow specifications to describe the behavior of one method relatively to the specification of another method passed as argument, and it must allow specifications to describe the behavior of a closure without exposing its captured state. This paper presents a modular specification and partial correctness verification methodology for closures. Our solution is based on first-order logic and, thus, well suited for automatic verification with SMT solvers. We present an encoding of our methodology in the Boogie program verifier. Using this encoding, we have verified a series of interesting examples that cover the main applications of closures such as delegation patterns and even the creation of custom control flow.

1 Introduction

A *closure* represents a first-class citizen procedure that is able to capture its lexical environment. Closures in an imperative language are a very powerful feature, since they can be used to succinctly implement important delegation-based design patterns such as Strategy and Command [12], to model objects, and even to create custom control flow.

Even though closures were introduced very early [28], they were ignored for a long time by mainstream object-oriented programming languages such as Java and C++. With their introduction into C# [10], Scala [26], and dynamic languages such as Python [2] and Ruby [11], closures now gain the popularity they deserve.

It can be argued that closures can be mimicked by objects, which have been studied extensively, and therefore the specification and verification of closures presents nothing new. This is however not so. Closures capture their lexical environment, something that objects do not automatically do. Furthermore, closures are much more concise and flexible than objects and therefore motivate a different style of programming. For example, implementing custom control structures is a common practice with closures. Such applications pose new challenges for the specification and verification.

Current research on closures [14, 15, 3, 16, 29, 24, 19] in imperative programming languages is using higher-order logic. When implemented, these methodologies use higher-order logic provers, such as Coq [13], to create proofs interactively. Our focus is different: we are interested in *automatic verification* using automatic first-order provers, in particular, SMT solvers such as Simplify [7] and Z3 [8]. SMT solvers are used in many modern program verifies such as ESC/Java [9], 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 first-order formalization of closures.

In this paper, we present a specification and partial correctness verification methodology for closures that is amenable to automatic verification. We show that our specification language is expressive enough not just to capture the two essential characteristics of closures (being first-class and capturing their environment), but also to deal elegantly with specifications that talk about other specifications as well as with custom control structures. The examples presented here, as well as other interesting examples, are already verified using the Boogie program verifier [22] and available online at www.pm.inf.ethz.ch/publications/closures.

The programming language used here only contains the features immediately relevant for closures. In particular, the only values we use are primitives and closures. Objects can be supported as a straightforward extension of the language with records, but we omit this feature for simplicity. We believe that other features of object-oriented programming such as inheritance, subtyping, and dynamic binding are orthogonal to closures and, therefore, omitted.

```
var f, g : () → Int , n, m : Int;
val counter =
  proc (x : Int) returns result : () → Int
  ⟨ var count : Int;
    val inc = proc () returns r : Int ⟨r := count ; count := count + 1⟩;
    count := x;
    call inc;   // A
    result := inc;
  ⟩;
f := call counter(40) ; call f ; n := call f;
assert n = 42;
g := call counter(99) ; m := call f ; n := call g ; n := n + m;
assert n = 143;
```

Fig. 1. Nested closures capturing state.

1.1 Introduction to Closures

The term "closure" is often used to refer to only those procedures that are used as first-class citizens, that is, passed as parameters to or returned from other procedures, or assigned to variables. We prefer a more uniform terminology: in a language that supports closures, *all procedures and side-effect free functions* are called closures, regardless of their use in the program.

The program in Fig. 1 demonstrates the basic features of closures. It implements a simple counter. First, a value *counter* is introduced. As in Scala, the keyword **val** indicates that *counter* is a constant: its value cannot change by an assignment. The right-hand side of the definition of *counter* is a *closure occurrence*, an expression that is introduced using the keyword **proc**.

The closure takes an integer parameter x. The return type of the closure is $() \to \mathsf{Int}$; the elements of this type are *closure instances* that take no parameters and return an integer. A program variable *result* is introduced to carry the return value. It is the final value of this variable that is returned to the caller.

In the body of the closure occurrence that we equated to constant *counter*, a local integer variable *count* is introduced. The variable *count* will hold the value of the counter at all times. A nested closure occurrence follows and it is equated to the constant *inc*. This introduces an instance of that closure, accessible through the name *inc*. Note that a *closure occurrence* is different from a *closure instance*: there may be many instances coming from one occurrence. In general, it is not known statically how many instances of an occurrence exist at a given state in a computation. In our example, each time *counter* is invoked, a new instance of *inc* (as well as a corresponding new instance of the variable *count*) is created. After the introduction of *inc*, the variable *count* is initialized to the formal parameter of the enclosing closure occurrence, x.

The closure instance *inc* that is created can access the variable *count*. The purpose of *inc* is to return the value of *count* and to increase it by 1. The closure instance *inc* is invoked once in its definitional environment (Line A), for demonstration purposes (the **call** operator denotes closure invocation).

Finally, the closure instance *inc* is assigned to variable *result*. This means that the closure instance *inc* will be returned by the closure instance *counter* to its caller. This demonstrates the first-class citizenship of closure instances; they can be assigned to variables and returned from other closure instances.

Now the main program follows: *counter* is invoked and variable f gets assigned a closure instance that increments a counter that begins at 41. We may invoke f, each time increasing the value of the counter. At the second invocation, we assign its value to n, and we see that n gets the value 42. This demonstrates closures capturing their environment. Although we are out of the lexical scope of variable *count*, the closure instance f still remembers it.

As we mentioned above, another invocation of *counter* will generate a new instance of *count* and therefore of *inc*. The rest of the code demonstrates that the two counters are indeed separate.

1.2 Approach and Contributions

To specify the behavior of closures, we equip every closure occurrence with a precondition, a postcondition, and a modifies-clause to declare potential heap modifications. While reasoning about such specifications is well understood for traditional procedures, closures lead to several challenges. In the following, we outline these challenges and our solutions.

Environments. The first challenge is to model the *state capturing* feature of closures. Closure instances carry with them a lexical environment, in which they execute. This environment is different from the local state that the fields of an object carry. In particular, unlike objects, many closure instances share (parts of) their lexical environments. For example, in Fig. 1, the closure instances f and g capture environments that partly overlap (they both agree on variable n for example, while they disagree on *count*). This observation suggests that the notion of the captured *environment* should play a prominent role in our state model. In Sect. 2, we introduce a state model, in which lexical environments are a runtime component of the state. Using this state model, we show the encoding in Boogie of the source programs, together with their standard pre/postcondition specifications.

Specification Functions. The Boogie encoding is straightforward for most programming constructs, but not for closure invocations. The problem with closure invocations is caused by the fact that closures are first-class citizens and therefore can be stored in variables, returned from other closures, and so on. This means that the invoked closure occurrence and, in particular, its specification may not be known statically, as illustrated by the invocations of f and g in Fig. 1.

Furthermore, the fact that a closure may take closures as parameters or return closures as results means that its specification must talk about these closures. In particular, the specification must talk about the *specifications* of its parameters and its results. This phenomenon occurs in our small example: the closure *counter* must ensure that the closure instance that it returns has a certain precondition (always true in this case) and a certain postcondition (it increments a variable by 1 and it returns its previous value).

To deal with both issues, we introduce *specification functions pre*, *post*, and others, whose job is to record the specification of a closure. Remarkably, the specification functions for a certain closure are defined by local assumptions, which encapsulates their behavior. On the other hand, these functions are known globally, which makes it possible to use them in the encoding of invocations of unknown closures occurrences and in specifications of closures.

A further application of specification functions is *state abstraction*. We introduce an optional abstraction specification function, locally defined, but known by the client. This deals with cases like our running example, where specifications need to express properties of a closure's captured state without referring to hidden variables such as *count*.

Dynamic Frames. The framing problem occurs in all modular programming methodologies. In Sect. 4, we show how we deal with the framing problem, by employing a simplified version of *dynamic frames* [17].

We then present another interesting example, which reflects a specification and verification challenge in [20], and which concerns *delegation*. Delegation involves working with unknown closures, the *delegates*. It is hard to decide even whether an invocation of a delegate is valid (that is, its precondition holds) at the place where it happens. We use specification functions and dynamic frames to solve the problem.

Custom Control Structures. In Sect. 5, we deal with a very powerful feature of closures: the creation of custom control flow. In particular, we present a custom while loop written using closures. The treatment of such very general constructs with pre/postconditions is hard, since loops can be used to achieve any satisfiable postcondition. To verify the loop and its clients, we use ghost parameters to pass specification-related information, in particular a loop invariant. The example verifies without introducing fixpoints or any higher-order logic constructs.

$$V ::= \ 0|1|...|\mathbf{true}|\mathbf{false}|\ \ [\mathbf{proc}\ (\overline{I:T})[\ \mathbf{returns}\ I:T]]\ \langle P\rangle \quad \text{values}$$

$$E ::= \ I \mid V \mid U\ E \mid E\ B\ E \mid (E) \quad \text{expressions}$$

$$T ::= \ \mathsf{Int} \mid \mathsf{Bool} \mid () \mid (\overline{T}) \to T \quad \text{types}$$

$$D ::= \ \mathbf{var}\ I:T \mid \mathbf{val}\ I = V \quad \text{declarations}$$

$$\begin{array}{rl} P_1 ::= & I := E \mid \mathbf{if}\ E\ \mathbf{then}\ (\ \overline{\overline{P_1}}\ ) \qquad \text{statements} \\ & \mid [I :=]\mathbf{call}\ E[(\ \overline{E}\ )] \mid \mathbf{assert}\ \ E \end{array}$$

$$P ::= \ \overline{\overline{D}}\ \overline{\overline{P_1}} \quad \text{programs}$$

Fig. 2. Programming Language. The only metasymbols in the rules are |, [], ::=, the overline, and the double-overline. The overline $\overline{\alpha}$ means zero or more comma separated occurrences of α. The double overline $\overline{\overline{\alpha}}$ means zero or more semicolon separated occurrences of α. Capital letters are non-terminal symbols. Non-terminal I denotes identifiers. Non-terminal U denotes primitive unary operators. Non-terminal B denotes primitive binary operators. Notice also that the **proc** and **returns** keywords are optional in closures: they can be omitted if there are no formal parameters and no return values, resp. We will introduce the syntax of specifications along with our methodology.

The syntax of our programming language is shown in Fig. 1.2. The semantics of the language should be clear after the discussion of Fig. 1. We will later introduce specification constructs as well as additional statements and expressions.

2 Environments

To handle state capturing, we introduce *environments*, that is, mappings from identifiers to locations. In this section, we present a state model that supports environments and show how we use environments to translate source code and specifications to BoogiePL, the input language of the Boogie verifier.

2.1 State Model

Closures are first-class citizen procedures. This means that the instances of closures are treated as values within the program. A first-order formalization would not permit storing the actual code of a closure as a value. Instead, each closure instance will contain a token that uniquely represents the closure occurrence from which it was created. We call this token the *key* of a closure occurrence. Keys are used to connect a closure instance to its code and specification.

A closure occurrence may create many different closure instances, which must be clearly separated from each other. For example, the program in Fig. 1 creates two closure instances out of the closure occurrence equated to *inc* and assigns them to variables f and g. What is different about these two instances is that each of them captures a *different* variable *count*. This means that *count* corresponds to a different location each time *counter* is called and that difference is significant since this location is captured and remembered by the corresponding instance of *inc*. This motivates the notion of *environment*, a mapping from identifiers to the actual locations that they represent in memory. The two instances of *inc* will share the same key, but they will carry with them two different environments.

Environments Env, closure instances CI, and values Val are defined as follows:

$$\begin{aligned} \mathsf{Env} &= [\mathsf{Ident}]\mathsf{Loc} \\ \mathsf{CI} &= \mathsf{CK} \times \mathsf{Env} \\ \mathsf{Val} &= \mathsf{Bool} \cup \mathsf{Int} \cup \mathsf{CI} \end{aligned}$$

where Ident is the set of identifiers, Loc is the set of locations (addresses), and CK is the set of closure keys. The BoogiePL notation $[X]Y$ represents a *map* from objects of type X to objects of type Y

Example 1. Consider the program in Fig. 1, and let u_C be the key of the closure occurrence that is equated to *counter* and u_I the key of the closure occurrence that is equated to *inc*. When executing the program, three closure instances are created: $(u_C; e_0)$, $(u_I; e_1)$, and $(u_I; e_2)$, for some environments e_0, e_1, e_2. The domains of the environments e_1 and e_2 are both equal to:

$$\{f, g, n, m, \mathit{counter}, \mathit{count}, \mathit{inc}\}$$

Environments e_1 and e_2 agree on identifiers f, g, n, m, and *counter*, but they do not agree on *count* and *inc*. □

Even though our language has no allocation and deallocation constructs, we need a heap to store the values of program variables that are captured by a closure and, therefore, still live even when their scope expires. Program variables that are not captured could be stored on the stack. However, we consider this to be a compiler optimization that we ignore; instead of performing a static analysis to determine which variables are being captured, we require that formal parameters and return variables are never captured (and, thus, live on the stack), whereas all local variables live on the heap.

The heap is modeled as a mapping from locations to values:

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

To obtain the value of a particular variable x, we first consult an environment e to obtain x's heap location and that a heap h to obtain the value stored in this location: $h[e[x]]$. The indirection via environments allows us in particular to model closure instances that share certain heap locations.

A closure occurrence C is a special case of expression, which is encoded as $(u; e)$, where u is the key of the closure occurrence and e is the current environment. We see here that a new closure instance is created, which captures the environment e in which the interpretation happens.

2.2 Encoding of Closure Occurrences

To verify a program, we need to prove that each closure occurrence satisfies its specification. For this purpose, we encode each closure occurrence together with its specification and body into a BoogiePL procedure. Boogie then generates verification conditions to prove that procedure implementations satisfy their specifications. Like closure occurrences, BoogiePL procedures are specified using pre/postconditions and a modifies clause.

In our state model, the state is a pair of a heap and an environment. We refer to the *current heap* by the global variable H. The environment of a closure instance is passed as an argument E of type Env to the BoogiePL procedure. This encoding of the state reflects that all executions of closure instances share one heap, but typically operate on different environments.

Example 2. Assume the following specification for the closure occurrence *inc* in Fig. 1 (for the time being, we omit all **modifies** specifications):

proc () **returns** $r : \mathsf{Int}$
  requires true
  ensures $r = \mathsf{old}(\mathit{count}) \;\wedge\; \mathit{count} = \mathsf{old}(\mathit{count}) + 1$
$\langle r := \mathit{count}\ ;\ \mathit{count} := \mathit{count} + 1 \rangle$

This gets translated into the following BoogiePL procedure:

```
procedure I(E : Env) returns (r : Int)
  requires true;
  modifies H, A;
  ensures r = old(H)[E[count]] ∧ H[E[count]] = old(H)[E[count]] + 1;
{
  r := H[E[count]] ; H[E[count]] := H[E[count]] + 1;
}
```

□

The encoding of statements in the source language is straightforward, as the statements of the source language have equivalent BoogiePL statements. Closure invocations are an exception, discussed in the next section.

For declarations, we use a global variable A whose type is

```
AllocFlag = [Loc]Bool
```

to ensure that newly-declared variables get assigned fresh locations. The source-language declaration,

```
var x : T;
```

gets translated to:

```
var newLocation : Loc;
assume ¬A[newLocation];
E[x] := newLocation;
A[newLocation] := true;
```

In the above code, we see a BoogiePL **assume** statement. Such statements introduce local assumptions for the current state, which are passed on to the prover. In the present example, the assumption guarantees that the location that is given to the identifier x is unallocated before the declaration.

Since no deallocation construct exists, we add to all procedures a *free* post-condition that deallocation did not happen during the execution:

```
free ensures ∀l ∈ Loc · old(A)[l] ⇒ A[l] ;
```

A free postcondition in BoogiePL may be assumed at the call site, but need not be proven for the procedure implementation.

3 Specification Functions

With the encoding presented in the previous section, we can encode invocations of closures whose closure occurrence is known statically by simple calls to the

corresponding BoogiePL procedures. In this section, we show how to encode invocations of arbitrary closure instances and how to write specifications that talk about other specifications.

We address both problems by introducing *specification functions* in the BoogiePL encoding. These functions record the specifications of closure instances. We use them to encode invocations of statically-unknown code and in closure specifications to refer to specifications of other closures.

3.1 Specification Functions Basics

Let us first present the technicalities behind specification functions and later show how these are used to deal with the two problems mentioned above.

The type of a closure instance has the form $(T_1, .., T_n) \to RT$, where the T_i are n (possibly zero) types of the formal parameters and RT is the result type (which can be () if there is no return value). For each such closure type T, we introduce globally two functions pre_T and $post_T$ with the following signatures:

$$pre_T \in \mathsf{CI} \times \mathsf{Heap} \times \mathsf{AllocFlag} \times T_1 \times .. \times T_n \to \mathsf{Bool}$$
$$post_T \in \mathsf{CI} \times \mathsf{Heap} \times \mathsf{AllocFlag} \times \mathsf{Heap} \times \mathsf{AllocFlag} \times T_1 \times .. \times T_n \times RT \to \mathsf{Bool}$$

Function pre_T takes a closure instance, the current heap and allocation information, and the actual arguments and yields whether the precondition of the closure holds. Function $post_T$ in addition takes the initial heap and allocation information, which are needed to encode $\mathsf{old}()$-expressions, and the return value, and yields whether the postcondition holds. Both functions are defined by *local assumptions* where a closure occurrence is declared. Therefore, even though all BoogiePL procedures know of the existence of the specification functions, they cannot use their definition for a specific closure instance directly in their reasoning. This serves the purpose of encapsulation.

Example 3. Consider the declaration of *inc* in Fig. 1 and its specification given in Ex. 2. Assuming that u_I is the key for the closure occurrence, the declaration is translated as follows:

```
// declaration of constant inc
var incLocation : Loc;
assume ¬A[incLocation];
E[inc] := incLocation;
A[incLocation] := true;

// equation of inc to an expression
H[E[inc]] := (u_I; E);

// axiomatization of closure occurrence
assume ∀h ∈ Heap, a ∈ AllocFlag, e ∈ Env ·  pre_{()→Int}((u_I; e), h, a)  =  true;
assume ∀oldh, h ∈ Heap, olda, a ∈ AllocFlag, r ∈ Int, e ∈ Env·
  post_{()→Int}((u_I; e), oldh, olda, h, a, r)  =
    (r = oldh[e[count]]  ∧  h[e[count]] = oldh[e[count]] + 1);
```

□

3.2 Encoding of Closure Invocations

In Fig. 1, we have an invocation of *inc* at line A. Since *inc* is a constant, we know that its value is precisely the closure instance $(u_I; e)$, and so we can encode the invocation by a call of the corresponding procedure, using BoogiePL's **call** statement, which generates a proof obligation for I's precondition (it *asserts* it), assigns arbitrary values to (*havocs*) all variables possibly modified by I, and then assumes I's postcondition:

$tmp := \textbf{call } I(\mathsf{E})\textbf{;}$

However, in general the closure occurrence to be invoked is not known statically such as in the invocations of f and g. They are variables and they are assigned the return value of invocations of *counter*. Such invocations are encoded using the specification functions. The encoding resembles the meaning of BoogiePL's **call** statement: first the precondition is asserted, then H and A are *havoced*, the postcondition and the free postcondition are assumed, and finally the result is assigned to the appropriate location.

Example 4. The encoding of

$n := \textbf{call } f\textbf{;}$

is:

$\textbf{assert } pre_{()\to\mathsf{Int}}(\mathsf{H}[\mathsf{E}[f]], \mathsf{H}, \mathsf{A})\textbf{;}$
$oldHeap := \mathsf{H}\ \textbf{;}\ oldAlloc := \mathsf{A}\ \textbf{;}\ \textbf{havoc } \mathsf{H}, \mathsf{A}, tmp\textbf{;}$
$\textbf{assume } post_{()\to\mathsf{Int}}(oldHeap[\mathsf{E}[f]], oldHeap, oldAlloc, \mathsf{H}, \mathsf{A}, tmp)\textbf{;}$
$\textbf{assume } \forall l \in \mathsf{Loc} \cdot \mathsf{old}(\mathsf{A})[l] \Rightarrow \mathsf{A}[l]\ \textbf{;}$
$\mathsf{H}[\mathsf{E}[n]] := tmp\textbf{;}$

where *tmp* is a fresh variable. □

3.3 Specifications About Specifications

Since the verification of closure invocations requires information about the closure instance's specification functions, this information has to be passed on from the place where the closure instance is created (and where the specification functions are defined by local assumptions) to all the places where the closure is invoked. This information passing requires that the specifications of closures that take or return closure instances need to express properties of the specification functions of those closure instances. For instance, the specification of the closure occurrence *counter* of Fig. 1 needs to express properties of its result closure such that clients of *counter* can invoke the result closure. To write such specifications, we allow the use of *pre* and *post* in specifications to refer to the specification functions (we omit the type subscript since we apply these functions to closure instances of statically known types).

Example 5. The following postcondition of *counter* expresses that the precondition of the result closure is always **true**, which allows clients to invoke the result closure. We will present a stronger specification for *counter* later.

$$\begin{array}{l}\textbf{val } \mathit{counter} \ = \textbf{proc } (x : \mathsf{Int}) \textbf{ returns } \mathit{result} : () \to \mathsf{Int} \\ \qquad\qquad\qquad\qquad \textbf{requires true} \\ \qquad\qquad\qquad\qquad \textbf{ensures } \forall h \in \mathsf{Heap}, a \in \mathsf{AllocFlag} \cdot \ \mathit{pre}(\mathit{result}, h, a)\end{array}$$

□

3.4 The Abstraction Specification Function

In Fig. 1, clients do not see the variables *count* of the closure instances f and g, but they still must be aware of f's and g's captured state because the result of invocations depends on it. To preserve information hiding, we provide clients with an *abstraction* of the state data. This section introduces state abstraction in our framework.

To support state abstraction, we extend the types of closures to also include an *abstraction type*, that is, the type of an abstract value of the state of the closure instance:

$$(\overline{T}) \to T[;T]$$

For every type

$$T = (T_1, .., T_n) \to RT; AT$$

we introduce a specification function abs_T to access the abstract value of a closure instance:

$$\mathit{abs}_T \ \in \ \mathsf{CI} \times \mathsf{Heap} \times \mathsf{AllocFlag} \times T_1 \times .. \times T_n \to AT$$

In the source language, the abstraction function is defined locally in the closure specification, using the special keyword **abstract** followed by an expression of the abstraction type. This introduces, like for the other specification functions, a local assumption that defines the function abs_T for that closure occurrence.

Example 6. In Fig. 1, we can use the abstraction function to abstract away variable *count*. The following **abstract** clause in the specification of *inc* expresses that the abstract value of a closure instance is the value of *count*. This is of course not a substantial abstraction, but it is good enough for our small example.

$$\begin{array}{l}\textbf{val } \mathit{inc} = \\ \ \ \textbf{proc } () \ \textbf{returns } r : \mathsf{Int} \\ \ \ \ \ \textbf{requires true} \\ \ \ \ \ \textbf{ensures } r = \mathsf{old}(\mathit{count}) \ \wedge \ \mathit{count} = \mathsf{old}(\mathit{count}) + 1 \\ \ \ \ \ \textbf{abstract } \mathit{count} \\ \ \ \langle ... \rangle\end{array}$$

The type of *inc* is now $() \to \mathsf{Int}; \mathsf{Int}$. A local assumption is introduced:

$$\textbf{assume } \forall e \in \mathsf{Env}, h \in \mathsf{Heap}, a \in \mathsf{AllocFlag} \cdot abs_{() \to \mathsf{Int};\mathsf{Int}}((u_I; e), h, a) \;=\; h[e[count]] \;;$$

□

In the source language, we can access the abstract value at the current heap using the **abs** operator.

Example 7. We can use **abs** to specify *counter*. Using the abstraction, the specification of *counter* can be more substantial than in Ex. 5:

```
val counter =
  proc (x : Int) returns result : () → Int
    requires true
    ensures ∀h ∈ Heap, a ∈ AllocFlag ·  pre(result, h, a)
    ensures abs(result) = x + 1
    ensures ∀oldh, h ∈ Heap, olda, a ∈ AllocFlag, r ∈ Int·
     post(result, oldh, olda, h, a, r)  ⇒
           r = (abs(result))_{oldh;E;olda}
       ∧ (abs(result))_{h;E;a} = (abs(result))_{oldh;E;olda} + 1
  ⟨...⟩;
f := call counter(40) ; call f ; n := call f;
assert n = 42;
```

In the above code, $E_{h;e;a}$ denotes the evaluation of expression E in heap h, environment e, and allocatedness map a.

The two new postconditions say (a) that the abstract value when the call returns is equal to $x + 1$, where x is the parameter of *counter*, and (b) the postcondition of the result closure ensures that the abstracted value will increase by one and that the return value of the invocation will equal the old abstract value. The code of *counter* is omitted. The part of the client that verifies with the new specification is also shown. □

3.5 Syntactic Sugar in the Specification Language

As we see in Ex. 6, the need to talk about various heaps and allocatedness maps, as well as the need to quantify over them complicated specifications. To alleviate the problem we introduce the following conventions:

- The current heap H and the old heap $\mathsf{old}(\mathsf{H})$, as well as the current and the old allocatedness map A and $\mathsf{old}(\mathsf{A})$, may appear in a specification
- Quantifications over these four variables are permitted. Such quantification hide the global variables with the same names. In particular a one-heap universal quantification will be abbreviated by $\forall_1$:

$$(\forall_1 \cdot E) \;=\; (\forall \mathsf{H} \in \mathsf{Heap}, \mathsf{A} \in \mathsf{AllocFlag} \cdot E)$$

and a two-heap universal quantification by $\forall_2$:

$$(\forall_2 \cdot E) \;=\; (\forall \mathsf{H}, \mathsf{old}(\mathsf{A}) \in \mathsf{Heap}, \mathsf{A}, \mathsf{old}(\mathsf{A}) \in \mathsf{AllocFlag} \cdot E)$$

– We introduce keywords **pre** and **post**, which always apply on these four variables, that is:

$$\mathbf{post}(f, v_1, .., v_n, r)$$

abbreviates

$$\mathit{post}(f, \mathsf{old}(\mathsf{H}), \mathsf{old}(\mathsf{A}), \mathsf{H}, \mathsf{A}, v_1, .., v_n, r)$$

and the same for **pre**

Example 8. With the syntactic sugar that we introduced, the specification in Ex. 6 becomes much more readable:

$$\begin{array}{l}
\mathbf{val}\ \mathit{counter}\ = \\
\quad \mathbf{proc}\ (x : \mathsf{Int})\ \mathbf{returns}\ \mathit{result} : () \to \mathsf{Int} \\
\quad\quad \mathbf{requires\ true} \\
\quad\quad \mathbf{ensures}\ (\forall_1 \cdot \mathbf{pre}(\mathit{result}))\ \wedge\ \mathbf{abs}(\mathit{result}) = x + 1 \\
\quad\quad \mathbf{ensures} \\
\quad\quad\quad \forall_2 \cdot \forall r \in \mathsf{Int} \cdot \mathbf{post}(\mathit{result}, r)\ \Rightarrow \\
\quad\quad\quad\quad r = \mathsf{old}(\mathbf{abs}(\mathit{result}))\ \ \wedge\ \ \mathbf{abs}(\mathit{result}) = \mathsf{old}(\mathbf{abs}(\mathit{result})) + 1
\end{array}$$

□

4 Dynamic Frames

With the machinery introduced so far, we addressed the two main challenges of closures: we can encode captured state via environments and express properties of this state via abstraction, and we can specify properties of closure instances via specification functions. The remaining issue is how to deal with the *frame problem*, that is, how to express which part of the heap is guaranteed not to be modified by a closure. For this purpose, we use a simplified version of *dynamic frames* [17]. In this section, we introduce the background on dynamic frames used in this paper and show how to apply the methodology to the example of Fig. 1 and an example for delegation.

4.1 Basics of Dynamic Frames

Dynamic frames specify the possible side-effects of a closure by providing for each closure a *frame*—a set of locations that serves as the "footprint" of the closure. No allocated location outside the frame must be modified by the closure. Even

though a client may not see the variables contained in the frame, it may refer to all of them using the frame as their abstract representation.

In this paper, we use a simplified version of dynamic frames, where the frame of a closure is constant, that is, does not depend on the heap. This restriction simplifies our examples, as framing, although essential, is not the main focus of the paper. An extension to heap-dependent frames is possible.

It is not important for the client to know the exact locations included in a frame. What is important, is to know that the frame is *disjoint* from other locations that are of interest to the client. If we know for example that the frame of a closure is disjoint from the address of a variable, then we are sure that the invocation of the closure will not change the value of that variable.

To formalize the above scheme, we need to express two concepts. First, we say that a frame F *frames* an expression E (denoted by F **frames** E) if the value of E *will not change* if all the locations of F maintain their values:

$$F \ \textbf{frames}\ E \ = \ \forall_2 \cdot (\forall l \in F \cdot \ \mathsf{old}(\mathsf{H})[l] = \mathsf{H}[l]) \ \Rightarrow \ \mathsf{old}(E) = E$$

Second, we say that a frame F is *respected* by a closure invocation, if the invocation does not modify any *allocated* location outside of F. The closure is allowed to allocate new locations and to modify them. Respecting a frame F (denoted by ΔF) is a two-heap specification expression formally defined as follows:

$$\Delta F \ = \ \forall l \in \mathsf{Loc} \cdot \neg \mathsf{old}(\mathsf{A})[l] \ \vee \ l \in F \ \vee \ \mathsf{H}[l] = \mathsf{old}(\mathsf{H})[l]$$

Now, we can say that if a closure C respects a frame F, a frame G is disjoint from F, and E is an expression framed by G, then the invocation of C will not change the value of E:

$$\Delta F \ \wedge \ F \cap G = \emptyset \ \wedge \ (G \ \textbf{frames}\ E) \quad \Rightarrow \quad E = \mathsf{old}(E)$$

This theorem is called the *framing theorem*. The client uses the framing theorem to ensure that invocations do not change important values that are unrelated to them. To make use of the framing theorem, we must show how each of the three conjuncts to the left of the implication are ensured.

Respecting the Frame. The frame of a closure occurrence is specified using a **modifies** clause, which indicates a set of locations. For example, the specification of *inc* in Fig. 1 should be given the following specification:

modifies $\{\mathsf{E}[\mathit{count}]\}$

which can be abbreviated by

modifies *count*

The frame of a specification is captured by yet another specification function, mod_T (for any closure type T). If $T = (T_1, .., T_n) \to RT$, then

$$mod_T \ \in \ \mathsf{Cl} \times T_1 \times .. \times T_n \to 2^{\mathsf{Loc}}$$

The mod_T specification function is defined with local assumptions, like the other specification functions.

To make sure that the closure respects the frame, we add ΔF to the post-condition of the corresponding BoogiePL procedure, and the translation of each invocation.

Example 9. The specification of *inc* includes the following modifies clause:

modifies $\{\mathsf{E}[count]\}$

This clause leads to the following postcondition in the BoogiePL procedure for the closure occurrence:

$$\textbf{ensures } \forall l \in \mathsf{Loc} \cdot \neg \mathsf{old}(\mathsf{A})[l] \ \vee \ l \in \{\mathsf{E}[count]\} \ \vee \ \mathsf{H}[l] = \mathsf{old}(\mathsf{H})[l] \ ;$$

which expresses that the closure modifies only locations that were not allocated in the pre-state or that are contained in the closure's frame. The modifies clause is also reflected by the local assumption for mod_T:

$$\textbf{assume } \forall e \in \mathsf{Env} \cdot \ mod_{() \to \mathsf{Int}}(u_I; e) \ = \ \{e[count]\} \ ;$$

where $(u_I; e)$ is the closure instance stored in *inc*. Finally, invocations of this closure instance include an additional assumption:

$$\textbf{assume } \forall l \in \mathsf{Loc} \cdot \neg \mathsf{old}(\mathsf{A})[l] \ \vee \ l \in mod_{() \to \mathsf{Int}} \ (\mathsf{H}[\mathsf{E}[f]]) \ \vee \ \mathsf{H}[l] = \mathsf{old}(\mathsf{H})[l] \ ;$$

□

When *counter* returns a result, the client does not know the variable *count*, but *counter*'s specification can use the specification function $mod_{() \to \mathsf{Int}}$, to refer to the frame of the closure instance that it returns. We use keyword **mod** in the source language to refer to any mod_T specification function.

Establishing and Maintaining Disjointness. In our simplified version of dynamic frames, frames do not vary with state, so maintaining frame disjointness is trivial. To *establish* the disjointness of existing frames with a newly created frame, we only need to follow two simple rules. First, any frame in use must be fully allocated, which is denoted by $alloc(F)$:

$$alloc(F) \ = \ \forall l \in F \cdot \ \mathsf{A}[l]$$

Second, a newly created frame must only contain *fresh* locations, that is, locations that were initially unallocated and are now allocated. This motivates the following notation, for the *freshness* specification:

$$\mathit{fresh}(F) \;=\; \forall l \in F \cdot\; \neg \mathsf{old}(\mathsf{A})[l] \;\wedge\; \mathsf{A}[l]$$

Given the fact that deallocation never happens, these two conventions guarantee that frames always remain disjoint.

The postcondition of *counter*, which we showed in Ex. 6 and 8, must be enriched with a conjunct that the frame of the new closure instance is fresh:

ensures *fresh*(**mod**(*result*))

This postcondition allows clients of *counter* to prove that the result closures of two invocations of *counter* operate on disjoint frames. In the main program from Fig. 1, this allows us to prove that invocations of f and g do not interfere.

Framing Expressions. The last conjunct that we need to apply the framing theorem is that a certain frame F *frames* an expression E. In our running example, we want the frame of the new closure instance to frame the evaluation of its abstraction function. We add the following conjunct to the postcondition of *counter*:

ensures mod(*result*) **frames abs**(*result*)

Verifying the Counter Example. All these specifications that we added to the running example make it possible to verify the whole code. The freshness specifications guarantee that the frames of f and g are disjoint. The **frames** specifications guarantee that the value of the abstraction of f does not change when g is invoked and vice versa. The example with all its assertions verifies. The complete encoding of the example can be found on www.pm.inf.ethz.ch/publications/closures (filename Counter.bpl).

4.2 A Delegation Example

Delegation based patterns (such as the Strategy, Chain of Commands, and Command design patterns [12]) are at the heart of programming with closures, as well as of object-orientation. Delegation means that a specific task is not carried out by the running method M, but passed on to another object, method, or closure D, the *delegate*. We then say that M *delegates* the task and we call it the *delegant*. The delegate is usually passed to the delegant as a parameter. This indirection achieves great flexibility, because there may be many delegates, carrying out the task in many different ways, even in (or especially in) ways that were not conceived at the creation time of the delegant.

The challenge that delegation causes is in the specification of the delegant [20]. Typically, the job of the delegant is not to achieve a certain postcondition, but to invoke the delegate. Specifying the delegant then, again, requires specifications about specifications, in this case, about the specification of the delegate. Moreover, it is even hard to determine whether the delegate is allowed to be invoked at a certain point (that is, whether its precondition holds). This is for instance the case when the precondition of the delegant requires the precondition of the delegate to hold, but the heap is modified before the delegate gets invoked and, thus, it is not obvious that the delegate's precondition still holds after the heap modification. Using specification functions and dynamic frames, we are able to address these challenges elegantly.

Example 10. In Fig. 3, we show an example, which represents the gist of a specification and verification challenge [20]. We have a closure f, which calls a delegate g (in Line A). We want to know that the precondition of g holds at that point. The global variable x, the local variable y, and the invocation of an unknown closure instance h serve to make the framing problem harder and more interesting.

```
var y : Int ; var h : () → () ;
val f = proc (g : () → ())
  ⟨ var x : Int ; x := 3 ; y := 4 ;
    call h ; call g ; // A
  ⟩;
```

Fig. 3. A Delegation Example.

The problem is how to specify f in such a way that it verifies, that is, such that the preconditions of all closure instances hold at the place of their invocation. This is achieved by a precondition for f that expresses the following properties:

- The preconditions of g and h initially hold.
- The frames of g and h *frame* their respective preconditions.
- The frames of g and h are fully allocated. This guarantees their disjointness from x, and therefore the fact that the assignment to x does not interfere with the preconditions of g and h.
- The location of y is not included in the frames of g and h. This guarantees that the assignment to y is non-interfering.
- The frames of g and h are disjoint, which guarantees that invoking h does not interfere with the precondition of g.

The following precondition summarizes all that formally:

$$\begin{array}{l} \quad \mathbf{pre}(g) \ \wedge \ \mathbf{pre}(h) \ \wedge \ \mathbf{mod}(g) \ \mathbf{frames} \ \mathbf{pre}(g) \ \wedge \ \mathbf{mod}(h) \ \mathbf{frames} \ \mathbf{pre}(h) \\ \wedge \ \mathit{alloc}(\mathbf{mod}(g) \cup \mathbf{mod}(h)) \ \wedge \ \mathsf{E}[y] \notin (\mathbf{mod}(g) \cup \mathbf{mod}(h)) \\ \wedge \ \mathbf{mod}(g) \cap \mathbf{mod}(h) \ = \ \emptyset \end{array}$$

Using that precondition, the example verifies. A subtle observation is the following: if we want f to not change the content of h, we also need to include

$$\mathsf{E}[h] \notin \mathbf{mod}(g) \cup \mathbf{mod}(h)$$

in the precondition. This example too can be found under the URL www.pm.inf.ethz.ch/publications/closures (filename Delegation.bpl). □

5 Custom Loops

In this section, we illustrate the expressiveness of our methodology by specifying and verifying the implementation of a while loop via closures. In this and many other examples, it is convenient to have *functional closures*, which are side-effect free and, thus, simplify framing and verification in general. For instance, the condition of our custom while loop can be expressed via a functional closure.

5.1 Functional Closures

A *functional closure* is a closure which returns a result but does not change the state. Like imperative closures, functional closures are first-class and they store their own environment, in which they are evaluated. Unlike imperative closures, their evaluation takes place in an expression and not in an invocation statement.

A functional closure occurrence is an expression written using the syntax $\lceil E \rceil$, where E is a programming language expression. The type of $\lceil E \rceil$ is written T_f where T is the type of E.

To evaluate a functional closure instance, the special keyword **eval** is used. An **eval** f expression retrieves the environment of closure instance f and evaluates f in the current heap and that environment. Notice the difference between **call** and **eval**: the former is a statement, whereas the latter is an expression and, in particular, may be used within specifications. For example, the following is a valid programming language expression:

$$\mathbf{eval} \ \lceil x + 1 \rceil + 2 - \mathbf{eval} \ g$$

Functional closures are treated very similarly to imperative closures. Their occurrences are given a unique key and their instances are identified by a key and their environment. For every functional type T_f, there is a specification function for the evaluation of the instance:

$$\mathit{eval}_T \ \in \mathsf{CI} \times \mathsf{Heap} \times \mathsf{AllocFlag} \to T$$

The definition of $eval_T$ happens with a local assumption at the place where the functional closure occurrence appears. The evaluation operator **eval** is encoded straightforwardly by a call to the respective $eval_T$ function.

For simplification of our mathematics, we require all functional closure instances to be *total*, that is, they may invoked in any heap. Preconditions for functional closures are analogous to preconditions for imperative closures. Moreover, our functional closures take no parameters but an extension is straightforward.

5.2 A Custom While Loop

The specification of the custom while loop differs significantly from the usual pre/postcondition style of specifications, since while loops have such a wide application span that they can be used to achieve any satisfiable postcondition. On the other hand, using induction and fixpoints to define such constructs does not serve our purposes, since it exceeds first-order logic.

To deal with the problem in a first-order setting, we use ghost parameters. *Ghost parameters* are specification-only parameters that do not appear in the final program, but are used only for reasoning. Using ghost parameters, we allow the clients to pass on specification-related information, such as loop invariants. Such information is needed by first-order theories to verify very general constructs like while loops. Ghost parameters appear to the end of the formal parameter list, and they are separated from the ordinary parameters with a semicolon.

Example 11. We implement and specify the custom while loop as follows:

$$
\begin{array}{l}
\textbf{val}\ \mathit{while} = \\
\quad \textbf{proc}\ (\mathit{condition} : \mathsf{Bool}_f\ ,\ \mathit{body} : () \to ()\ ;\ \mathit{inv} : \mathsf{Bool}_f) \\
\quad\quad \textbf{requires}\quad \textbf{eval}\ \mathit{inv} \\
\quad\quad\quad\quad \wedge\, \forall_2 \cdot \mathsf{old}(\textbf{eval}\ \mathit{condition}\ \wedge\ \textbf{eval}\ \mathit{inv})\ \wedge\ \textbf{spec}(\mathit{body})\ \ \Rightarrow\ \textbf{eval}\ \ \mathit{inv} \\
\quad\quad \textbf{modifies}\ \textbf{mod}(\mathit{body}) \\
\quad\quad \textbf{ensures}\ \neg\textbf{eval}\ \mathit{condition}\ \wedge\ \textbf{eval}\ \mathit{inv} \\
\quad\quad \langle \textbf{if}\ \textbf{eval}\ \mathit{condition}\ \textbf{then}\ (\textbf{call}\ \mathit{body}\ \textbf{;}\ \textbf{call}\ \mathit{while}(\mathit{condition}, \mathit{body}; \mathit{inv}))\rangle\ \textbf{;}
\end{array}
$$

where $\textbf{spec}(S)$ is an abbreviation of

$$\mathsf{old}(\textbf{pre}(S))\ \ \Rightarrow\ \ \textbf{post}(S)\ \wedge\ \Delta(\textbf{mod}(S))$$

The specification expresses the usual partial correctness conditions for while loops: The invariant has to hold before the loop and after every iteration. In our methodology, the client of *while* passes the invariant as a functional closure, and proves that it initially holds and that it is preserved by the body of the loop. The implementer of *while* promises that 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. This is an oversimplification that comes from the fact that frames do not change with state.

If we had the full dynamic frames theory, we would also pass a frame to the loop as a ghost parameter.

The above specification allows clients to reason about invocations of the loop. For instance, we can verify the following code:

```
var i, j, x : Int;
i := 0 ; j := 5 ; x := 7;
call while ( ⌈i < 5⌉ ,
             proc () modifies i, j ensures i = old(i) + 1 ∧ j = old(j) − 1
               ⟨i := i + 1 ; j := j − 1⟩ ;
             ⌈i + j = 5 ∧ i ≤ 5⌉);
assert i = 5 ∧ j = 0 ∧ x = 7;
```

The need to give a specification to the imperative closure passed as loop body makes the specification somewhat verbose. We expect that at least for simple examples, such specifications can be inferred.

The while example can be found online under the URL www.pm.inf.ethz.ch/publications/closures (filename While.bpl). □

6 Soundness

The encoding of our methodology in BoogiePL does not introduce any axioms. Therefore, the only potential source of unsoundness are the local assumptions that we add for each closure occurrence and for each closure invocation. However, these assumptions correspond exactly to the specifications of the BoogiePL procedures generated for each closure occurrence. Therefore, the verification of these procedures justifies the assumptions by showing that they are feasible. Therefore, the only situation that might lead to unsoundness is circular reasoning, that is, if an assumption was used to verify the code that is supposed to justify the assumption. In this section, we restrict closure specifications to prevent such circular reasoning and guarantee soundness of the encoding.

Circular reasoning potentially occurs when a specification function is defined in terms of itself. Consider the following example:

```
var x : () → ();
val f = proc () requires pre(x) modifies mod(x) ensures post(x) ⟨call x;⟩;
x := f;
```

All works well: f only calls x and it is reasonable to adopt its specification exactly. However, x and f have the same type, which means that x could equal f (see the assignment that follows the closure occurrence). This means that the definition of specification functions of f is circular! In particular, after the assignment $x := f$, our local assumptions imply that the precondition of f is equal to the precondition of x, which is equal to the precondition of f, etc. While this example does not cause unsoundness, it illustrates that unanticipated recursion through the heap is potentially dangerous. The following example, regardless of

the body of f, is actually unsound because in heaps in which $x = f$, we have $\mathbf{pre}(f) = \neg\mathbf{pre}(f)$:

var $x : () \to ()$**;**
val $f =$ **proc** () **requires** $\neg\mathbf{pre}(x)$ $\langle ... \rangle$**;**
$x := f$**;**

To prevent circular definitions, we introduce an important restriction. This restriction ensures that all closure types are ordered, and that the specification functions of a closure occurrence may be defined only in terms of specification functions that are smaller in that ordering. The requirement restricts specifications, but does not prevent recursive closure implementations.

To establish an ordering, we define the *size* of a type of the programming language as follows:

- The size of Int and Bool and () is 1.
- The size of T_f is the size of T plus 1.
- The size of $(T_1, .., T_n) \to RT$ is equal to the sum of sizes of the T_i and of RT plus 1.

We restrict our specifications as follows: any specification function (pre_T, mod_T, $post_T$, abs_T, and $eval_T$) may be defined only in terms of specification functions of types U whose size is strictly less that T.

This ensures non-circularity of the definitions, and therefore prevents the unsoundness. Notice that all our previous examples respect this restriction, while the example of the present section does not: the types of f and x are equal. Notice also that the restriction is not unreasonably strong: it allows a specification to talk about the specification functions of the formal parameters and the result value.

To avoid any misinterpretation, we should stress that the above restriction *does not forbid recursion*. It forbids the circular definitions of specification functions, that may result by (probably unwanted and unanticipated) recursions through the heap. In particular, the following recursive definition, which does not use specification functions, is permitted:

var i : Int**;**
val $f =$ **proc** () **requires true modifies** i **ensures** $i = 0$
$\langle$**if** $i > 0$ **then** $(i := i - 1$ **; call** f**;**$)\rangle$

Note that if i starts off negative, then we have non-termination. This is not a problem for a partial correctness framework such as ours.

Going back to the problematic example, consider the following variation:

var i : Int **; var** $x : () \to ()$**;**
val $f =$ **proc** ()$\langle$**call** x**;**$\rangle$ **;** $x := \langle i := 0$**;**$\rangle$ **;**
call f **; assert** $i = 0$**;**

It is unfortunate that we cannot verify this code, since the specification of f must find a way to reason about the specification of x, which is not permitted anymore. A way to deal with this issue is to attach specifications to closure variables. If variable x is constrainted by a specification S, an assignment $x := y$ should *assert that the closure instance y respects S in all heaps.* For example, the following example attaches a specification to x:

```
var i : Int;
var x : () → ()  requires true modifies i ensures i = 0;
val f = proc () requires true modifies i ensures i = 0 ⟨call x;⟩ ;
x := ⟨i := 0;⟩;  // A
call f;
assert i = 0;
```

Note that the code would still verify if we changed the assignment in Line A to $x := f$, even though that would induce non-termination. Again, partial correctness allows this to happen. On the other hand, the following assignment in Line A would not verify:

$$x := \langle i := 1; \rangle$$

because it does not satisfy the assertion that the assigned closure instance respects the specification of x.

7 Related Work

The most comprehensive framework for dealing with imperative higher-order language features is that of Yoshida, Honda and Berger [14, 15, 3, 16, 29], a total correctness Hoare-logic based framework. While it covers all technical aspects of closures, 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.

Closures are also treated in *Hoare Type Theory* [24], 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 [25] 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 [4] has been employed to deal with closures. An exploration of higher order frame rules appears in [5], but the language does not involve first-class closures and the modularity is static. A stronger language is used in [19], where several design patterns are elegantly formalized. The main idiom used in that formalization is delegation; there is no discussion on state capturing, specification changes, and custom control flow.

Currently, one of the best solutions for delegation-based patterns is JML's *model programs* [27]. Model programs are specifications that are written in the style of abstract programs. The problem with this approach is that it sometimes generates too strong specifications. In the present paper, we deal with delegation using only abstraction mechanisms, such as the specification functions. This approach typically generates weaker specifications, mainly dealing with framing issues. Ex. 10 shows that interesting problems posed by delegation can be solved this way. However, it is interesting, and one of our goals, to extend our methodology with support similar to model programs.

The Ex. 10 is the running example of [23], where we use closure invariants to guarantee properties that need to hold whenever a closure is invoked. However, maintaining these invariants is complicated. Other challenges of closures such as captured state and specifications about specifications are not addressed in that paper.

8 Conclusion

We have introduced, step by step, a specification and verification methodology for closures. Our solution is modular and designed with automated verification in mind. Modularity is achieved by verifying each closure separately. Automated verification with SMT solvers is enabled by a first-order formalization. We have encoded and verified all the examples presented in this paper in the Boogie program verifier.

Our approach orchestrates a lot of formalism. We showed a state model with environments, to model state capturing. We introduced specification functions that make it possible to encode arbitrary invocations and to have specifications about other specifications. We used an abstraction specification function for state abstraction. We used dynamic frames to deal with the framing problem. Finally, we showed how to use ghost parameters and functional closures to verify custom control flow closures.

Closures have many more applications than the ones shown here. Among other things, they are employed to mimic aspects [18], in the generation of native and non-native code (as in .NET LinQ), in patterns involving self-changing code and in delegation based patterns, such as Strategy, Command, etc. [12]. We plan to extend our specification methodology to treat applications like the above, especially targetting the delegation problem. For the treatment of delegation, we plan to extend our methodology with *model programs* [27]. We also plan to use the full theory of dynamic frames, in which more interesting examples can be handled.

Acknowledgments. We are grateful to Gary Leavens for an interesting discussion and to the anonymous referees of a predecessor of this paper.

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-Verlang, 2004.
2. D. M. Beazly. *Python Essential Reference*. SAMS, 3rd edition, 2006.
3. M. Berger, K. Honda, and N. Yoshida. A logical analysis of aliasing for higher-order imperative functions. In *ICFP05*, pages 280–293, 2005.
4. 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.
5. 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.
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, May 2005.
8. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *Lecture Notes In Computer Science*, pages 337–340. Springer-Verlang, 2008.
9. D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998.
10. ECMA. C# language specification. Technical Report Standard 334, ECMA, 2006.
11. D. Flanagan and Y. Matsumoto. *The Ruby Programming Language*. O'Reilly, 2008.
12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.
13. N. A. Hamid. Theorem proving with the COQ proof assistant: Tutorial presentation. *Journal of Computing Sciences in Colleges*, 24(2), 2008.
14. K. Honda. From process logic to program logic. In *ICFP04*, pages 163–17. ACM, 2004.
15. K. Honda and N. Yoshida. A compositional logic for polymorphic higher-order functions. In *PPDP04*, pages 191–202. ACM, 2004.
16. K. Honda, N. Yoshida, and M. Berger. An observationally complete program logic for imperative higher-order functions. In *LICS05*, pages 270–279, 2005.
17. 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-Verlang, 2006.
18. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, L. C. Videira Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *ECOOP'97*, volume 1241 of *Lecture Notes In Computer Science*, pages 220–242. Springer-Verlang, 1997.

19. N. Krishnaswami, J. Aldrich, L. Birkedal, K. Svendsen, and A. Buisse. Design patterns in separation logic. In *TLDI '09*, pages 105–116. ACM, 2009.
20. 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.
21. K. R. M. Leino. *Toward Reliable Modular Programs*. PhD thesis, Caltech University, 1995.
22. K. R. M. Leino. This is Boogie 2. Working Draft - available at `http://-research.microsoft.com/en-us/um/people/leino/papers.html`, 2008.
23. P. Müller and J. Ruskiewicz. A modular verification methodology for C# delegates. In U. Glässer and J.-R. Abrial, editors, *Rigorous Methods for Software Construction and Analysis*, 2007. To appear.
24. A. Nanevski, G. Morrisett, and L. Birkedal. Hoare Type Theory, polymorphism and separation. *Journal of Functional Programming*, 18(5-6):865–911, 2008.
25. A. Nanevski, G. Morrisett, L. Birkedal, A. Shinnar, Paul Govereau, and L. Birkedal. Ynot: Reasoning with the awkward squad. In *ICFP '08*, 2008.
26. M. Odersky, L. Spoon, and B. Veners. *Programming in Scala*. Artima, 2007.
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. G. J. Sussman and G. L. Steele Jr. Scheme: An interpreter for Extended Lambda Calculus. Technical Report AI Lab Memo AIM-349, MIT AI Lab, 1975.
29. N. Yoshida, K. Honda, and M. Berger. Logical reasoning for higher-order functions with local state. *CoRR*, abs/0806.2448, 2008.