Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Implementing Closures in Dafny

Research Project Report

- **Author:**
  Alexandru Dima [1]

- **Total number of pages:**
  22
- **Date:**
  Tuesday 28th September, 2010
- **Location:**
  Zürich, Switzerland

[1] *E-mail: adima@student.ethz.ch*

Contents

1 Introduction

Closures represent a particularly useful language feature. They provide a means to keep the functionality linked together with state, providing a source of expressiveness, conciseness and, when used correctly, give programmers a sense of freedom that few other language features do.

Smalltalk's standard control structures, including branches (if/then/else) and loops (while and for) are very good examples of using closures, as *closures delay evaluation*; the state they *capture* may be used as a private communication channel between multiple closures closed over the same environment; closures may be used for handling User Interface events; the possibilities are endless.

Although they have been used for decades, static verification has not yet tackled the problems which appear when trying to reason modularly about closures. This project is a proof of concept of the methodology defined in [1], which presents a modular specification and partial correctness verification methodology for closures.

The work described in this report is the result of a Computer Science Research Project at ETH Zurich, under the supervision of Dr. Ioannis Kassios and Prof. Dr. Peter Müller.

2 Background

2.1 Closures

A closure is a first-class imperative procedure with free variables that are bound in its lexical environment. It is defined within the scope of its free variables, and the extent of those variables is at least as long as the lifetime of the closure itself.

One way to think of a closure is as a package of two items: a pointer to a function and a pointer to a state, a set of variables-values bindings. Closures can be typically handled as any other programming language object, they can be passed as parameters, they can be stored to variables, and so on.

Figure 1, written in JavaScript, shows a simple example of an adder factory. The procedure *createAdder* will return a closure that will *remember a*, even after the execution has left the body of *createAdder*. More specifically, as long as a reference to x exists, a will also exist, and it will have the value with which the outer procedure was called.

```
function createAdder(a) {
  return function(b) {
    return a + b;
  }
}
x = createAdder(4);
y = createAdder(-1);
c = x(6);   // c ==  4 +  6 == 10
d = y(11);  // d == -1 + 11 == 10
```

Figure 1: Example of JavaScript closures.

What is happening in this specific example is that a new scope object containing all the local variables is being created every time *createAdder* is being executed and it is initialized with any variables passed in as procedure parameters. When *createAdder* finishes execution that scope is not destroyed because the returned procedure still holds an implicit pointer to it.

This mechanism allows x to be called and it can run using a, even though execution has left the body of *createAdder*.

2.2 Dafny

Dafny[2] is an experimental language for dynamic-frames specification[3]. It explores the dynamic frames style specifications in an object-based sequential setting.

Frame conditions define which allocated objects a method is allowed to change during its execution. They allow reasoning about the state of the heap after a method executes, as objects not listed in the *modifies* clause of the method remain unchanged.

The granularity of the Dafny frame conditions is restricted to objects. For example, if an object o has two integer fields, $o.i1$ and $o.i2$, in order to change $o.i1$, a method must list o in its *modifies* clause, because integers are basic types and not objects. This means that even if the method only wants to change $o.i1$, after the method call, unless ensured through an *ensures* clause that $o.i2$ is unchanged, no assumptions can be made about the state of $o.i2$ relative to its state before the call.

Dynamic Frames[3] allow to have framing without affecting data abstraction. In Dafny, this allows the programmer to list in its modifies clause a reference to a set of objects, set which *may change* from call to call, depending on the overall state of the heap and the purposes the programmer has in mind.

Figure 2 outlines the process of verifying a program written in Dafny. First, the Dafny program is parsed, an abstract syntax tree is built and resolved. Only if both the parsing and the resolving finished successfully, the Dafny abstract syntax tree is translated into a BoogiePL[4] abstract syntax tree. Then, if the resulting Boogie representation is correct, it is further translated into Math Formulae which are passed on to a SMT Solver (such as Z3[5]).

The results from the SMT Solver are collected and the possible error traces are translated back into Boogie tokens, which in turn are translated back into Dafny tokens, in order to present Dafny programmers with reasonable error messages.

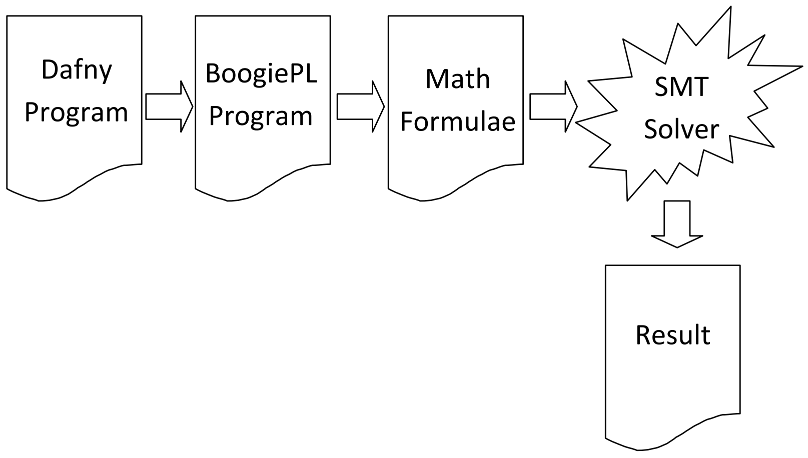

Figure 2: Dafny Verification Process

3 General approach

The implementation of closures in Dafny follows closely the methodology described in [1]. Closures are separated in two distinct types: *procedural closures* and *pure closures*. The differences and similarities between these two types are almost the same as those between methods and functions in Dafny.

Pure closures may be viewed as a special case of procedural closures which are guaranteed to have no side effects, which have a statically empty modifies clause. They can just evaluate an expression on a heap without actually making any changes to it. Therefore, the only operation one can do with a pure closure is to evaluate it. Since they are closures and do capture their environment, we have decided to make pure closures parameter free.

On the other hand, procedural closures may have side effects, meaning they may modify objects on the heap, and in general have a lot more properties than pure closures. They may have input parameters, output parameters, frame properties and an abstraction. The abstraction is used as a means to argue about the captured state of the procedural closures.

Like in [1], a number of *closure specification functions* have been introduced to the specification language. They are used to argue about closures and for a procedural closures they are:

- *pre(Heap,closure,inputs):boolean* – evaluate if the precondition of a closure holds on a given heap with given inputs.
- *mod(closure,inputs):set<Object>* – returns the static set of objects the closure may modify – its frame property.
- *abs(Heap,closure):AbstractionType* – returns the abstraction of a closure.
- *post(preHeap,postHeap,closure,inputs,outputs):boolean* – evaluate if the postcondition of a closure holds on a pre-heap and a post-heap, with the given inputs and outputs.
- *spec(preHeap,postHeap,closure,inputs,outputs):boolean* – evaluate that the closure acts like a method – given a pre-heap and a post-heap, the inputs and the outputs evaluates: "if the precondition holds on the pre-heap and we call the closure, will the frame property and the postcondition hold on the post-heap?".

For pure closures:

- *eval(Heap,closure):Type* – evaluates the pure closure's expression and returns the result

These functions, as defined above in BoogiePL are made available to the Dafny programmer. However, heaps may not be referred directly from Dafny, therefore the heap parameters to the functions will be filled in based on the context they are used from. The functions will be explained in greater depth in the following sections. Another newly introduced language construct is:

- *X frames Y*, where *X*'s type is *set<Object>* and *Y* usually is *pre(..)* or *abs(..)* – it returns a boolean and evaluates: "Is it true that if I change anything but objects from *X*, *Y* will remain constant?".

Another important change is in the semantics of the already existing Dafny language construct, *fresh(X)*, which until now evaluated "Is it true that *X* was not allocated in the pre-heap?" and now evaluates "Is it true that X is allocated in the post-heap and was not allocated in the pre-heap?".

The only difference from [1] is around working with a closure's context (environment). Our approach is to statically mark methods which define closures and closure definitions which define inner closures and extract an environment class for each one. This environment class will contain all the scope variables of the method and all usage of the local variables inside the method's body are substituted with indirections through this environment class, where each local variable has a corresponding field.

Another deviation is that the procedural closures bodies are being extracted and verified outside the method where the closure definitions statically occured, using again the environment class constructed for that method in order to indirect all references of scope variables.

In my opinion, the best way to understand how closures work in Dafny is to follow the examples from the following sections.

4 Procedural Closures

One of the most important additions to Dafny are the procedural closures. They resemble Dafny's methods in the sense that they may have side-effects (i.e. executing a procedural closure may change the heap) and they enable the use of statically unknown code in method calls.

Procedural closures, once created, may be passed around as parameters, returned by methods and generally behave like usual objects. The most notable difference to an object is that procedural closures may be called (in the same manner a method can be called) and by using some newly added specification functions, the programmer may use some characteristics of the closure to argue about its behavior.

4.1 Procedural Closure Type

Procedural closures are type-safe. This means that their type is statically known at compile-time, and type safety can be assumed by the Dafny programmer. The declaration for a procedural closure type is similar to the declaration of a method, because it lists the input types and the returned types. The only conceptual difference is that procedural closures might define abstractions of their captured state. The grammar for the type is:

([*Type* {, *Type*}]) -> ([*Type* {, *Type*}]) : ([*Type*])

Figure 3 shows examples of different procedural closure types. At this point, it is not important to understand what the abstraction means, but rather that a procedural closure may have one or more inputs, may have one or more outputs and may or may not be abstracted by a type.

Type	Explanation
() -> () : ()	procedural closure which takes no parameters, doesn't return anything and has no abstraction defined.
(*int*) -> () : ()	procedural closure which takes an integer as an input parameter, doesn't return anything and has no abstraction defined.
(*int*) -> (*bool*) : ()	procedural closure which takes an integer as an input parameter, returns a boolean and has no abstraction defined.
(*int*) -> (*bool*) : (*int*)	procedural closure which takes an integer as an input parameter, returns a boolean and is abstracted by an integer.

Figure 3: Examples of procedural closure types.

4.2 Procedural Closure Specifications

When defining procedural closures, it is possible to specify the behavior of these closures, in a similar manner to defining the behavior of methods. For a closure, these specifications include the precondition, the postcondition, the frame property and its abstraction. Because procedural closures encapsulate state, the abstraction is a means to expose state information.

4.3 A basic procedural closure example

Figure 4 shows a simple example of a procedural closure definition and usage.

4.3.1 Discussion

The closure *f*, which takes an integer, *n*, and returns the sum *n+5*, is declared like any other program variable and is defined like a method. Line 7 shows how procedural closures can be passed around as a reference (similar to usual objects); From lines 8 onwards, *g* will have the same state and the same behavior as *f*. Line 9 shows how a procedural closure can be called (like usual Dafny methods).

```
 1.  var f, g : (int)->(int):();
 2.  f := method (n:int)->(r:int):()
 3.    ensures r == n + 5;
 4.  {
 5.    r := n + 5;
 6.  };
 7.  g := f;

 8.  var y : int;
 9.  call y := g(5);  // y := 5 + 5
10.  assert y == 10;

11.  assert (forall1 n:int:: pre(current(f),n));
12.  assert (forall2 n,r:int::
          post(current(f),n,r) ==> r == n + 5);
```

Figure 4: Basic usage of procedural closures.

Line 11 shows how it is possible to argue about a closure's precondition. It expresses that for all heaps and input parameters, the precondition of *f* holds. The first newly-introduced language construct used is the *forall1::X* which expresses that for all heaps, *X* holds. It also modifies the context of translation for *X*, changing the current heap expression.

In the following sections it will be explained that, for this example, *f* resides on the heap, therefore the change in the context will also influence *f.* Because our intent is to argue about the current value of *f*, we must use yet another newly introduced language construct, *current(Y).*

Line 12 expresses that calling the procedural closure *f* will result in the return value being equal to the sum of the input parameter and 5. In this case, we use *forall2::Y*, expressing that for all pre-heaps and post-heaps, *Y* holds. Again, we need to use *current()* to reset the translation context to the current heap.

Also note how the built-in closure specification functions, *pre* and *post* are used. *pre* takes the procedural closure as the first parameter and as subsequent parameters all the input parameters to the closure. Similarly, *post* takes the procedural closure as first parameter, after which all the input parameters, and also all the returned values of the closure. Moreover, *post* may only be used in the context of arguing about two heaps (e.g. the *forall2* construct).

4.3.2 Boogie output

Although this is a basic and simple example, it is interesting to see and understand how the newly introduced Dafny concepts are translated into BoogiePL. Note that in this section, the machine-generated code has been modified to make it human readable and that some parts of it have been left out for brevity. Also, feel free to skip this section if you are only interested in understanding how to use the new Dafny constructs and not in understanding how the underlying Boogie code is generated.

Closure Type Class Figure 5 shows the generated class for the only closure type that appears in the Dafny source code: (int)->(int):(). In this context, the word class may be misleading because Boogie does not support classes, but the closure type declaration in Boogie is similar to any other Dafny class declaration translated to Boogie.

The closure type's class is being generated and in this case it is *Closure#1*, because it is the first closure type that appears in the source code. In general, for each unique closure type, one such class is generated.

The most interesting generated Boogie constructs are the closure specification functions. For our type, (int)->(int):(), which has one input parameter, an integer, and returns an integer, the *pre* specification function takes a heap, the closure instance and one integer and returns a boolean. In general, the *pre* function takes a heap, the closure instance, and the list of input parameters.

The *post* specification function takes two heaps (the pre-heap and post-heap), the closure instance, one integer, *f0*, corresponding to the input parameter and one more integer, *f1*, corresponding to the return value and returns a boolean. In general, the *post* function takes the pre-heap, the post-heap, the closure instance, the list of input parameters and the list of returns.

The *mod* specification function does not depend on the heap, therefore it is static (i.e. it is not possible to have a modifies clause which lists a variable set) and depends on the closure instance and the input parameters. The reason it depends on the input parameters is because of the possibility to pass closures as parameters, and in order to execute them, the programmer must add their modifies clause to his own. The function *mod* returns a set of objects, and in general depends only on the closure instance and the input parameters.

The *spec* specification function corresponds to the one defined in [1] and it is always defined with the axiom presented in Figure 5, adapted of course for the variable input and output parameters. The function *spec* states that a given closure acts as a method over two heaps, it expresses that if the preconditions hold on the pre-heap, calling it will result in its postconditions and the frame

```
// class.Closure#1 refers to all closures of type
// (int)->(int):() -- both f and g in our example
const unique class.Closure#1: ClassName;

function Closure#1.pre(H: HeapType,
              this: ref, f0: int) : bool;

function Closure#1.post(oldH: HeapType, H: HeapType,
              this: ref, f0: int, f1: int) : bool;

function Closure#1.spec(oldH: HeapType, H: HeapType,
              this: ref, f0: int, f1: int) : bool;

function Closure#1.mod(this: ref, f0: int) : Set BoxType;

axiom
(forall oldH:HeapType, H:HeapType, this:ref, f0:int, f1:int ::
  Closure#1.spec(oldH, H, this, f0, f1)
  <==>
  Closure#1.pre(oldH, this, f0) ==>
    Closure#1.post(oldH, H, this, f0, f1) &&
    (forall<alpha> o: ref, f: Field alpha ::
      o != null && oldH[o,alloc] ==>
        H[o,f] == oldH[o,f] || Closure#1.mod(this, f0)[$Box(o)]
    )
);
```

Figure 5: The generated closure type class for (int)->(int):().

property.

One missing specification function from this example is *abs*, because (int)->(int):() is not abstracted by anything. However, in general, the *abs* specification function depends on the heap and the closure instance and returns an object of the same type it was declared as.

Environment Class Figure 6 shows the generated environment class. Once again, it is not a Boogie class per se, but it is similar to how any Dafny class gets translated into Boogie. The translator creates one environment class per method containing at least one closure definition. In this example, the local variables *f*, *g* and *y* have been extracted and are now available as fields of this class.

```
const unique class.Env#1: ClassName;
const unique Env#1.f: Field ref;
const unique Env#1.g: Field ref;
const unique Env#1.y: Field int;
```

Figure 6: The generated environment for the simple example.

Main program Figure 7 shows how the simple example is translated into Boogie, line by line. In order to understand what is going on, it is required to mention that the Dafny Heap is represented as a Boogie map with two indices (a 2 dimensional array – the reference to an object and the fully-qualified name of that field). The heap is always available as the global variable *$Heap*.

Also, this excerpt does not show the initialization of the environment variable, *$env*, which is straightforward. Besides the environment, three new local variables have been generated, one to hold the heap during closure calls, *$oldH*, one used when defining the closure, *closure*, and another one used to fetch the return value of a closure call, *k*.

It is important to keep in mind that when translating the closure definition on lines 2-6 from Figure 4, the translator only uses the information available in the specification (i.e. not the body on lines 4-6). Also, the definition of the closure is created upon the local variable *closure*, in order to allow the local variable f to be reusable. If the definition would be created directly on f it would result in a contradiction when assigning a different value to it.

Because f has no precondition, *pre(f)* is defined to be true for all heaps and all input parameters. Also, because it does not modify anything, *mod(f)* is defined to be the empty set. *post(f)* is straightforward as it is equivalent to the written ensures clause on line 3. Finally, after these properties have been defined on the non-reusable local variable *closure*, it is assigned to *f*. Please also note that all the original variables, *f*, g and y are always indirected through the environment, to their respective fields.

The closure call on line 9 is also relevant to our discussion. First, the current heap is stored into *$oldH*, then the precondition of g is asserted with the input value 5. Next, the current heap is havoced. Then, the postcondition of g is assumed with the input value 5 and the output to the non-reusable local variable *k*. The last operation is the assumption of the frame property, which, in layman's terms states that all the existing objects from the old heap have the same state in the new heap or they belong to g's modifies clause. Finally, y is assigned the value of k – this is once again to allow y to be reused.

```
var $oldH: HeapType;
var $env: ref;
var closure: ref;
var k: int;

// --- Lines 2-6 --- f’s definition
assume (forall H:HeapType, n:int ::
    Closure#1.pre(H, closure, n) <==> true);
assume (forall H:HeapType, n:int ::
    Closure#1.mod(closure, n) == Set#Empty():Set BoxType);
assume (forall oldH:HeapType, H:HeapType, n:int, r:int ::
    Closure#1.post(oldH, H, closure, n, r) <==> r == n + 5);
$Heap[$env, Env#1.f] := closure;

// --- Line 7 --- g := f;
$Heap[$env, Env#1.g] := $Heap[$env, Env#1.f];

// --- Line 9 --- call y := g(5);
$oldH := $Heap;
assert Closure#1.pre($oldH, $oldH[$env,Env#1.g], 5);
havoc $Heap;
assume Closure#1.post($oldH, $Heap, $oldH[$env,Env#1.g], 5, k);
assume (forall<alpha> o: ref, f: Field alpha ::
    o != null && $oldH[o,alloc] ==> $Heap[o,f] == $oldH[o,f] ||
      Closure#1.mod($oldH[$env, Env#1.g], 5)[$Box(o)]);
$Heap[$env, Env#1.y] := k;

// --- Line 10 --- assert y == 10;
assert $Heap[$env#6, Env#1.y] == 10;

// --- Line 11 ---
assert (forall n:int, H:HeapType ::
    Closure#1.pre(H, $Heap[$env, Env#1.f], n));

// --- Line 12 ---
assert (forall n:int, r:int, oldH:HeapType, H:HeapType ::
    Closure#1.post(oldH, H, $Heap[$env, Env#1.f], n, r) ==>
      r == n + 5);
}
```

Figure 7: The generated main source.

The translation of line 11 shows how the implicit heap is added to the *forall1* language construct and how *pre* is invoked with that implicit heap. It also shows how *current(f)* gets translated to *$Heap*[*$env,Env#1.f*]. If *current()* is not used, *f* will be translated to *H*[*$env,Env#1.f*].

Finally, the translation of line 12 shows how the two implicit heaps are added to the *forall2* language construct and how *post* is invoked with them.

Closure's implementation The last missing piece of the puzzle is shown in Figure 8 and it is the closure's implementation. This is verified in a different procedure, showing the modularity of this approach. This procedure has an additional parameter, the environment of the method the closure was defined in and its specification is the same as the specification of the closure definition, plus the requirement that the environment parameter is not null (in case the closure would use an environmental variable), and the modifies clause for the *$Heap*, in case the procedure would change the heap, together with the frame property which in this case states that all objects in the pre-heap will remain the same in the post-heap.

```
procedure Closure#1.impl#1(this: ref, $scope_env: ref, n: int)
    returns (r: int)
  requires $scope_env != null;
  modifies $Heap;
  ensures r == n + 5;
  free ensures (forall<alpha> o: ref, f: Field alpha ::
        o != null && old($Heap)[o, alloc] ==>
           $Heap[o,f] == old($Heap)[o,f]);
{
  var self: ref;

  assume (forall H:HeapType, $n:int ::
    Closure#1.pre(H, self, $n) <==> true);
  assume (forall H:HeapType, $n:int ::
    Closure#1.mod(self, $n) == Set#Empty():Set BoxType);
  assume (forall oldH:HeapType, H:HeapType, $n:int, $r:int ::
    Closure#1.post(oldH, H, self, $n, $r) <==> $r == $n + 5);

  r := n + 5;
}
```

Figure 8: The generated closure's implementation.

The first part of the closure implementation is the definition of the *self* variable which is the same as the definition of the actual closure itself from Figure 7. This variable is defined in case the closure is recursive (i.e. it calls itself) and in that case the call in the Boogie source is made to the *self* variable. The assumption made by using this *self* variable is that a procedural closure does not redefine itself inside its own implementation.

Finally, the actual body of the closure's definition follows, which is a simple assignment.

4.4 A counter factory example

This example is adapted from [1]. The closure *counterF(x)* is a counter factory, it creates procedural closures which generate consecutive numbers starting at the input parameter, x. The first time one of these counters is called, it will return x, the second time, it will return *x+1*, the third time, *x+2*, and so on.

An important point in this example is that the created counter closures are independent from one another, the creation of a counter closure does not interfere with the existing closures and interleaving counter closures calls does not interfere with their consecutive numbers generation. This is a strong cue that in this example the counters are each created with their own private state.

In Figure 9 a procedural closure named *counterF* is declared on line 4. From its type, we can see that it takes one input parameter, an integer, and returns a procedural closure. The return type is a procedural closure with no input parameters, which returns an integer and is abstracted by an integer.

The rule of thumb when writing procedural closures specifications is to imagine that the body of the procedural closure is not visible to the clients (the places where the closure is called from) and to think of the body merely as an implementation for the closure's specification. In this case, even though the source code reveals the closure's implementation, we must realize that the lines 11 through 23 (*counterF*'s implementation) are not visible to the lines 24 through 33 (*counterF*'s client).

Therefore, we must rely completely on the specifications to verify a procedural closure's clients. This is the reason why the specification of *counterF* (lines 6 through 10) looks so bulky and expresses so many things. Let's review each of the ensures clauses and understand what they express and why they are needed for this example:

a) `ensures (forall1:: pre(c));`

   This ensures that for all possible heaps, the precondition of c (the returned counter) will hold. In layman's terms, it means that c may be called

```
class IntWrapper {
  var v : int;
}

var counterF : (int)->( ()->(int):(int) ):();
counterF := method (x:int)->( c:()->(int):(int) ):()
  ensures (forall1:: pre(c));
  ensures (forall2 r:int:: post(c,r) ==>
     abs(c)==old(abs(c))+1 && r==old(abs(c)));
  ensures mod(c) frames abs(c);
  ensures abs(c) == x;
  ensures fresh(mod(c));
{
  var cnt := new IntWrapper;
  cnt.v := x;
  c := method ()->(result:int):(int)
    requires cnt != null;
    modifies cnt;
    abstracts cnt.v;
    ensures cnt.v==old(cnt.v)+1 && result==old(cnt.v);
  {
    result := cnt.v;
    cnt.v := cnt.v + 1;
  };
};

var f, g : ()->(int):(int);
call f := counterF(40); // abs(f) == 40
var n, m : int;
call n := f();          // n == 40 && abs(f) == 41
call n := f();          // n == 41 && abs(f) == 42
call g := counterF(99); // abs(g) == 99
call m := f();          // m == 42 && abs(f) == 43
call n := g();          // n == 99 && abs(g) == 100
n := n + m;
assert n == 141;
```

Figure 9: Example of a counter factory using procedural closures.

anytime without any restrictions after its creation. Note that in this case the specification function *pre* is called with only one parameter, the closure *c*, because of *c*'s type (c:()->(int):(int)), which takes no input parameters.

b) `ensures (forall2 r:int:: post(c,r) ==>`
```
              abs(c)==old(abs(c))+1 && r==old(abs(c)));
```

   This ensures that for all possible pre-heaps and post-heaps, calling *c* and assigning the returned value to *r* will result in *c*'s abstraction in the post-heap being the sum of *c*'s abstraction in the pre-heap and 1 and that the returned value *r* will be *c*'s abstraction in the pre-heap. This clause is responsible for stating the closure's behavior (i.e. what calling the closure returns and how the abstraction of its state changes to reflect the call). In this case it shows how calling this closure will return the state's abstraction and how the state's abstraction will be incremented.

c) `ensures mod(c) frames abs(c);`

   This clause states that the abstraction of *c*'s state depends only on objects in it's modifies clause. This means that unless the programmer modifies the objects in *c*'s modifies clause, *c*'s abstraction will remain constant. This is one of the properties needed to express that multiple closures do not interfere with each other (another way to think of it is that their state depends only on the objects they modify).

d) `ensures abs(c) == x;`

   This expresses that after creation, the returned counter closure *c* will be initialized in such a manner that its state's abstraction is equal to *counterF*'s input parameter, *x*. This, combined with b) shows that when first called, *c* will return *x*, and that the second call will return *x*+1, and so on.

e) `ensures fresh(mod(c));`

   The final missing piece of the puzzle and the second of the two properties needed to prove that multiple closures do not interfere with each other is that the newly created counter *c* will modify only newly introduced objects in the heap, relative to the client's perspective. Together with c) which expresses that the counter's state only depends on what it modifies, stating that what it modifies is fresh, results in the property that multiple counter closures do not interfere in any way with each other.

After understanding *counterF*'s specification, we can now observe that it is complete enough in order to allow the proving of the client on lines 24 to 33 without any knowledge of how *counterF* is implemented. Therefore, this is a source of modularity, *counterF*'s implementation may be verified independently of its clients, a fact which has been addressed during the implementation of the new Dafny features.

Let's discuss *counterF*'s client on lines 24 - 33. Keeping *counterF*'s specification in mind, we can go over the client line by line. On line 25, a counter closure is created and assigned to local variable *f*, once again showing how closures may be passed around as references. At this point, we should keep in mind that *f* may be called at any time without any constraints, that it is initialized at 40 and that it relies on objects newly allocated.

Calling f twice on lines 27 and 28 will result in predictable behavior, it will first return 40, the value it was initialized with, and increment it's abstraction to 41 and then return 41 and increment its abstraction to 42. On line 29, another counter closure is created and assigned to local variable g, which is initialized at 99 and with a behavior that relies on objects once again newly allocated. This means that g does not depend on objects that counter f does.

The last lines of the client are straightforward, because creating g did not change any objects that f depends on (notice that *counterF* does not have any modifies clause, meaning it does not modify any existing objects on the heap), f's state is the same as before *counterF* was called and on line 30 it will set m to 42 and increment its abstraction to 43. Calling g on line 31 will result in n having the value 99. On line 32, n will become the sum of 42 and 99, which will obviously yield a correct assertion on line 33.

Now that we have seen that the client can verify with *counterF*'s specification, let's see how it has actually been implemented. A notable behavior in lines 12-22 is that the definition of the returned closure c will capture the environment in which it was declared (i.e. *counterF*'s body). That is why it is possible for both c's specification and implementation to use the variable *cnt* introduced on line 12. The environment and more specifically the variable *cnt* will survive on the Heap even after execution leaves *counterF*'s body and will be tied together with c.

Variable *cnt* is allocated on line 12 which helps proving e) and is initialized with x on line 13, which helps proving d). The specification of the returned closure c is once again sufficient to prove *closureF*'s body, even without using c's body on lines 19 - 22. c requires that the local variable *cnt* has been allocated, it expresses that it modifies it and that it is abstracted by its value.

The trick around creating an IntWrapper class comes from Dafny's intrinsic requirements that a modifies statement is coarse-grained to the level of objects – a programmer cannot use object's fields of basic type in such a modifies clause (i.e. in the modifies on line 16, one can write *cnt*, but not *cnt.v* because *cnt.v* is an integer).

Line 17 shows how a closure can link an expression based on its state to its abstraction. In this case, it is almost a one to one mapping (c's state is *cnt*, while its abstraction is *cnt.v*). c's specification ensures that when calling c, *cnt.v* will be incremented and that the return value, *result* will be the old value of *cnt.v*.

Lines 16 and 17 help prove the specification c), because c is abstracted by *cnt.v* and it modifies *cnt* and because it is obvious that *cnt* frames *cnt.v* (i.e. if *cnt* is not changed, *cnt.v* is also not changed).

Since *c* is abstracted by *cnt.v*, it is obvious how line 18 will help prove the specification b). Since *c* only requires that *cnt* is allocated and since *c* captures *cnt* immediately upon its definition and since *cnt* is allocated on line 12, this precondition will hold for all heaps (*cnt* will be captured after *c*'s definition, and as long as one can argue about *c* in any future heap, *cnt* will implicitly exist). Therefore, a) will also hold.

Now that we have seen how *counterF*'s body implements correctly its specifications, the only judgment left to be carried out is to verify that *c*'s body also implements correctly its specifications. This is trivial, as it is obvious how lines 20-21 correctly implement *c*'s specification: *result* will be the old *cnt.v* and *cnt.v* will be incremented.

4.5 Delegation example

This example is adapted from [1]. It shows the expressiveness of the newly introduced language constructs in order to argue about completely unknown closures.

In Figure 10, a procedural closure *f* is defined. It is using its environment, accessing the environmental variable *y* and calling the environmental closure *h*. Also, it is using a procedural closure, *g*, given as an input parameter.

The example's main point is to show what specifications a Dafny programmer must write in order to be able to call the two closures and how to argue about their frames and their interference.

The preconditions on lines 8 and 9 are straightforward as they express the fact that both *g* and *h* must be callable once. (After calling, it is not necessary that the precondition still holds). Similar to the example in Figure 9, we require that the objects listed in the modifies clauses of *g* and *h* frame their preconditions (lines 10 and 11).

What allows us to call *g* on line 22 after calling *h* on line 21 is the information that calling *h* does not affect in any way the precondition of *g*. This is assured with the requires on line 12, which states that their modifies sets are disjoint, meaning that when we will call *h*, it may change objects in *mod(h)*, but it will definitely not change any objects in *mod(g)*, which frames *pre(g)*, therefore *pre(g)* will still hold.

The last three preconditions (lines 13-15) refer to *y*, asking that *y* is not in the modifies sets of the two closures, *h* and *g*. This will also imply that changing *y* on line 20 will not influence the possibility to call the two closures (their preconditions). The modifies clause on line 16 contains *y* and also the modifies

```
class IntWrapper {
  var v : int;
}

var y := new IntWrapper;
var h : ()->():();
var f : ( ()->():() )->():();

f := method ( g: ()->():() )->():()
  requires pre(g);
  requires pre(h);
  requires mod(g) frames pre(g);
  requires mod(h) frames pre(h);
  requires mod(g) !! mod(h);
  requires !(y in mod(g));
  requires !(y in mod(h));
  requires y != null;
  modifies mod(g), mod(h), y;
{
  var x : int;
  x := 3;
  y.v := 4;
  call h();
  call g();
};
```

Figure 10: Delegation example using procedural closures.

sets of the two closures which are being called from *f*. This is required because if *f* would not list these as objects being modified, its specification would be incorrect.

The actual body of the closure is straight-forward and its successful verification proves that the reasoning was correct and that the calls are allowed.

5 Pure Closures

5.1 Pure Closure Type

Pure closures are also type-safe, their type is statically known at compile-time, and type safety can be assumed by the Dafny programmer. The declaration for a pure closure type is similar to the declaration of a variable, because it contains its type. The grammar for the type is:

< *Type* >

Figure 11 shows examples of different pure closure types.

Type	Explanation
< **bool** >	pure closure type which evaluates to a boolean.
< **int** >	pure closure type which evaluates to an integer.

Figure 11: Examples of pure closure types.

5.2 A recursive while

This example is adapted from [1]. In Figure 12, an implementation for the familiar *while* language construct is shown, using both procedural and pure closures.

```
 1.  var while : ( <bool>, ()->():(), <bool> )->():();
     // c - condition, b - body, i - invariant
 2.  while := method ( c:<bool>, b:()->():(), i:<bool> )->():()
 3.    requires eval(i);
 4.    requires (forall2::
          old(eval(c)&&eval(i)) && spec(b) ==> eval(i));
 5.    requires (forall1:: eval(c) && eval(i) ==> pre(b));
 6.    modifies mod(b);
 7.    ensures !eval(c) && eval(i);
 8.  {
 9.    if( eval(c) )
10.    {
11.      call b();
12.      call while(c, b, i);
13.    }
14.  };
```

Figure 12: A recursive while using closures.

Our *while* closure, defined on line 2, takes a boolean pure closure representing the continuation condition, a procedural closure which would represent the repeated body and another boolean pure closure representing the loop invariant, which is used only in the specification (as a ghost parameter).

Like usual while constructs, the procedural closure defined requires that the invariant holds upon entry (line 3), that executing the body when the condition holds, maintains the invariant (line 4) and that the body can be executed when the condition holds (line 5).

The *while* closure modifies whatever the body *b* modifies when it is called (line 6) and it ensures that the condition will no longer hold and the invariant will (line 7), just like an usual while construct.

The implementation for the *while* closure is a recursive one, which executes the body and calls itself if the continuation condition holds and returns immediately otherwise. It is straightforward to see and verify how the body implements the specification.

```
 1.  var i := new IntWrapper;
 2.  var j := new IntWrapper;
 3.  var x := new IntWrapper;
 4.  i.v:=0;  j.v:=5;  x.v:=7;

 5.  var condition : <bool>;
 6.  condition := ‘ i.v < 5 ‘;

 7.  var invariant : <bool>;
 8.  invariant := ‘ i.v + j.v == 5 && i.v <= 5 ‘;

 9.  var body : ()->():();
10.  body := method ()->():()
11.    requires i != null && j != null && i != j;
12.    modifies i, j;
13.    ensures i.v == old(i.v) + 1;
14.    ensures j.v == old(j.v) - 1;
15.  {
16.    i.v := i.v + 1;
17.    j.v := j.v - 1;
18.  };

19.  call while ( condition, body, invariant);
20.  assert i.v == 5 && j.v == 0 && x.v == 7;
```

Figure 13: Client example for the recursive while closure.

In Figure 13, we have a client implementation for the previously defined *while* closure. In this case, a while loop would have been less verbose, but this example aims to prove the expressiveness of the new Dafny concepts.

It is easy to see how this client increments i from 0 to 5, while decrementing j from 5 to 0, such that *invariant* holds at each iteration of the loop. Also, the continuation condition is written to stop incrementing i after 5. The assertion on line 20 will hold and proves once more how expressive this approach is.

6 Conclusions

6.1 Limitations

Due to time constraints (150 hrs in theory for the entire work) and the complexity of the problem, we did not have enough time for testing. Therefore, besides these known issues, some other problems may exist:

- *this* cannot be used from closures. However, it can probably be assigned to a local variable before the closure definition – not tested, though;
- the *decreases* language construct hasn't been implemented for closures. This means that the Dafny programmer may define the precondition for a closure f to depend on the precondition of another closure g, which might depend on f's precondition, and so on..., without receiving a verification error.

6.2 Possible extensions

We believe the work we did may be used as a base for adding closures to Dafny. Of course, some features are missing and some bugs may arise, but we are fairly confident in the overall quality of the code. Moreover, the code we added only runs when closures are used, so it does not interfere with Dafny programs not using closures.

The *decreases* language construct, the possibility of using *this* inside closure definitions and dynamic frames would be good extensions.

6.3 Acknowledgments

I would like to thank my supervisor Dr. Ioannis Kassios and Prof. Dr. Peter Müller for their support and a pleasant collaboration and Dr. K. Rustan M. Leino for his work on Dafny and for his prompt and useful response when I contacted him.

References

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

[2] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. *16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-16)*, 2010.

[3] I. T. Kassios. Dynamic Frames: Support for Framing, Dependencies and Sharing without Restrictions. *Formal Methods*, 4085 of Lecture Notes in Computer Science:268–283, 2006.

[4] Robert DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical report, Microsoft Research, 2005.

[5] Leonardo De Moura and Nikolaj Bjorner. Z3: An efficient SMT solver. *In Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.