Adding Closure Support to Chalice

Prateek Agarwal

Master's thesis report

Chair of Programming Methodology
Department of Computer Science
ETH, Zurich

http://www.pm.inf.ethz.ch
November 2010

Supervisors:
Dr. Yannis Kassios
Prof. Dr. Peter Mueller

Contents

Listings

List of Tables

Abstract

Proving correctness of programs is a challenge in computer science research. The problem gets harder if the programming language contains closure, an important feature in object oriented languages. The goal of this thesis is to come up with a methodology to verify concurrent programs in the presence of closures.

Chalice is a language and program verifier for modular verification of programs. We have chosen to extend Chalice because it supports concurrency with object orientation but it has no support for closures. To achieve our goal we identify significant challenges in adding closure support to Chalice. We propose new syntax constructs to use closures in the language and present methodology for verification of newly introduced syntax elements.

We have extended Chalice program verifier and provided a test suite of Chalice programs to cover all the aspects of the extended language.

Chapter 1

Introduction

1.1 Motivation

In programming methodology research, special languages are created with minimal features to focus on core concepts of the language in isolation. These languages contribute as proof of concept to demonstrate the efficacy of a certain programming methodology. However to realize their potential the languages have to be developed on par with languages used in industry such as C#, Java etc. One of the biggest hurdle in this path is for different language features to work together.

1.2 Adding Closure support to Chalice

Chalice[3][4] is an object based programming language. It provides methodology to formally verify that programs are free from concurrency issues such as deadlocks and race conditions. Chalice has minimal support for object oriented features. One such feature heavily used in OO languages is closure. Closures are methods which can capture their lexical environment. Using closures one can write concise programs and utilize newer programming patterns. With their introduction in C# [9] and Scala [10] and dynamic languages such as Python [11] and Ruby [12] closures have gained popularity in main stream industrial usage. Recently a first order formalism has been developed for verification of sequential programs with closures [1].

The goal of this thesis is to use ideas from [1] to add closure support in Chalice. The Chalice implementation has been extended with new features to support verification of programs with closures. The new methodology uses specification functions and abstract functions as described by Kassios and Mueller[1] with modifications.

1.3 Overview

This document presents the syntax and framework extension to the Chalice language. In Chapter 2 relevant features of existing language are discussed in brief. For details of the programming language and underlying verification methodology one should refer to [3] and [4]. Chapter 3 describes the major challenges in adding closure support to existing Chalice framework. Chapter 4 describes the newly introduced syntax elements to solve the challenges. Detailed examples are presented in Chapter 5. Chapter 6 describes the the methodology used for verification and shows formal encoding for proof obligations. Chapter 7 presents the conclusion of the work.

Chapter 2

Background in Chalice

This chapter describes features of syntax and verification framework of Chalice which are relevant to understand the extension provided later in the document. The Chalice program verifier works by translating Chalice programs into equivalent BoogiePL programs. BoogiePL is an intermediate language intended for program analysis and verification. BoogiePL programs can be verified automatically using the Boogie program verifier [7] and an SMT solver like Z3 [8] without any human assistance.

2.1 Chalice Verification Methodology

Chalice uses modular verification by using explicit method specifications. The method caller is concerned only with the method contract visible to it. A method contract includes the types of formal arguments, returned values and method specifications. The obligations for a method call is to establish method's preconditions before the method call. Its postconditions can be safely assumed after the method call terminates.

Methods in their specifications indicate access permissions required and ensured by them on shared state of program they use. Each potentially shared memory location is associated with access permissions to read or write.

2.2 Permissions

Chalice uses Fractional Permissions proposed by Boyland [6]. A permission is a fraction ranging between 0 and 100% inclusive. No permission to read or write is expressed as 0% permission and full permission to read as well as write by 100%. Any fraction in between 0 and 100% (exclusive) implies only read access. The syntax: acc(o.f,m) is used to indicate m% access of the field f of object o and rd(o.f) implies a positive, infinitesimally small permission ϵ and rd(o.f,n) means n units of ϵ. acc(o.f) is equivalent of acc(o.f,100) and rd(o.f) is equivalent to rd(o.f,1)

At the point of object creation the creating thread is given full access of the object and all its fields. In Listing 2.1. When a method is called, permissions are transferred to the called method based on the called method's precondition. After the method terminates it gives back the permissions present in its postconditions. Since the method *upCount* updates the field count, it has to specify full access in its precondition whereas the method *hasReached* requires only a read access to compare the field value with the input parameter. Both methods release the acquired access permissions in their post conditions. If *upCount* had not declared *acc(count)* in its postcondition, the call *c.hasReached(1)* would not be successful.

```
class Counter {
  var count:int

  method upCount()
    requires acc(count)
    ensures  acc(count) {
    count := count + 1
  }

  method hasReached(num:int) returns (b:bool)
    requires rd(count)
    ensures  rd(count) {
    b := ( count == num )
  }

  method main() {
    var c := new Counter
    call c.upCount()
    var b := call c.hasReached(1)
    assert(b)
  }

}
```

Listing 2.1: Example with read and write access

2.3 Threads

Threads in Chalice are created by the fork statement using a method, indicating parallel execution of the method in a new thread. Using the join statement a thread can wait for another thread to complete and get the result(s) of the method.

```
fork tok1 := c.upCount()     // invoke thread 1
join tok1                    // wait for thread 1 to terminate
fork tok2 := c.hasReached(1) // invoke thread 2
join b := tok2               // wait for thread 2 to terminate
```

Listing 2.2: Parallel Threads in Chalice

Listing 2.2 shows an alternative to invocation of the methods *upCount* and *hasReached* shown in Listing 2.1. Token variable *tok1* is used to identify the instance of newly created thread. A token can be used by the creating thread to wait upon the invoked thread's termination by using join statement. To fork a thread, the preconditions specified by the method should be satisfied and all access permissions in the preconditions are taken away. After the invoked thread terminates, its result can be obtained by the caller in join statement and method's postconditions are ensured to be satisfied at this point.

A parallel invocation of methods *upCount* and *hasReached* should fail because they access the same shared state that is the field *count*. Verification of following lines would fail at the second statement. When a new thread is created using method *upCount* the executing thread gives away 100% of the field *count* to newly created thread. At the second statement, the executing thread is left with 0% whereas method *hasReached* requires 1 unit of ϵ permission.

```
fork tok1 := c.upCount()
fork tok2 := c.hasReached(1)
```

Permission transfer between method call happens through two procedures known as Exhale and Inhale. Exhale(E) checks if current thread owns the required access permissions inside the expression E and that E holds. After this the access permissions are taken away. Inhale(E) is the inverse of E that is E is assumed and all access permissions in E are given to the current thread. When a thread is invoked the preconditions are exhaled by the invoking thread and they are inhaled back only after the thread joins. Hence method call is equivalent to a fork and join in subsequent statement. Following two pieces of code are equivalent:

```
call retvals := foo(inputs)
```

```
fork tk := foo(inputs)
join retvals := tk          //equivalent to call
```

2.4 Functions

Functions are a mechanism in Chalice to abstract over values of expressions. In Listing 2.3 the class *Account* uses the function *isPositive* to yield if the field *amount* is positive or not. In order to evaluate it, the calling thread should have access of the *amount*. This is specified in the precondition of the function. Since functions are side effect free they can't alter access permissions or program state, hence they require only preconditions. Intuitively this is equivalent of having the same postcondition as precondition

```
class Account {
  var amount:int

  function isPositive(): bool
   requires acc(amount)
  { amount > 0 }

  method add(increment:int)
    requires acc(amount) && isPositive() && increment > 0
    ensures  acc(amount) && isPositive()
  {
    amount := amount + increment
  }
}

class Banker {

  method process() {
    var clientAccount := new Account
    clientAccount.amount := 1
    call clientAccount.add(10)
    assert(clientAccount. isPositive ())
    }

}
```

Listing 2.3: Chalice functions for abstraction

2.5 Predicates

Predicates are a mechanism to abstract values of expressions as well as permissions. When permissions inside definition of a predicate are to be abstracted, the predicate is folded which means all access permissions in predicate's definition are taken away from the calling thread and 100% permissions are transferred to predicate. Similarily when a predicate is unfolded, if the thread has access of the predicate then all access permissions inside the definition of the predicates are given to the thread.

The *Account* class given in Listing 2.3 can be extended using predicates to hide the access permission of the field *amount* as shown in Listing 2.4. Within the body of method *add* the statement *unfold hasAccess* verifies because access of *hasAccess* is mentioned in the method's precondition. After the unfold statement, access of the predicate is taken away and access of the field *amount* is given to the thread. The statement *fold amount* takes away the access of the field and access of the predicate *hasAccess* are given back. If the field *amount* is updated either before the unfold statement or after the fold statement, the program would not successfully verify.

```
class Account {
    var amount:int

    predicate hasAccess { acc(amount) }

    function isPositive(): bool
     requires hasAccess
    { unfolding hasAccess in amount > 0 }

    method add(increment:int)
      requires hasAccess && isPositive() && increment > 0
      ensures  hasAccess && isPositive()
    {
      unfold hasAccess
      amount := amount + increment
      fold hasAccess
    }

    method create()
      requires acc(amount)
      ensures  hasAccess && isPositive()
    {
      amount := 1
      fold hasAccess
    }
}

class Banker {

  method proces() {

    var clientAccount := new Account
    call clientAccount.create()
    call clientAccount.add(10)
    assert(clientAccount.isPositive())

  }
```

```
}
```

Listing 2.4: Predicates

2.6 Program State

The state of a Chalice program at any point of execution is represented by a heap, permission mask and local variables. Local variables are declared in stack as their values are needed inside the method body. However this needs to be changed in the presence of closures.

All objects in Chalice are allocated on a global heap which is a 2 dimensional map which maps object reference and field names to their values.

Each permission is tuple of two integers: R, N to denote the fractional permission: $R + N \cdot \epsilon$. Permissions of all memory locations are stored in a 2 dimensional map called PermissionMask which maps object reference and field names to fractional permission corresponding to the memory location. Integer values are used instead of fractions because many popular SMT solvers have limitations on use of rational numbers.

Chapter 3

Challenges in adding closure support

This chapter shows a sample program in Section 3.1 which uses closures. Sections 3.2,3.3 and 3.4 illustrate the significant challenges associated with verification of concurrent programs in the presence of closures using the sample program. Different elements of the newly introduced syntax are discussed in detail in Chapter 4.

3.1 Sample Program

The program in Listing 3.1 implements a method *makeCounter* which creates two closures: *up* and *down* by assigning anonymous methods to them. These closures operate on variable *count* which is local to the method *makeCounter*. Since both the closures access the shared memory location, *count* they specify *acc(count)* in their pre/post conditions. The client method calls the method *makeCounter* to associate instance of the returned closure with variables u and d. It can then invoke these closures using variables u and d in sequence or in parallel.

```
class ClosureExample {

method makeCounter()
        returns (up:()-->int , down:()-->int)
 {

   var count := 0

   up := method() returns count : int
           requires acc(count)
           ensures  acc(count) && count = old(count) + 1
         {
           count := count + 1
         }

   down := method() returns count : int
             requires acc(count)
             ensures  acc(count) && count = old(count) - 1
           {
             count := count - 1
           }
}

method client() {
  var i : int
```

```
    var j : int
    var u : () --> (int)
    var d : () --> (int)
    var u1 : () --> (int)
    var d1 : () --> (int)

    call u,d := call makeCounter

    call i := u();
    call j := d();

    fork u()
    fork d()

    call i := u1();
    call j := d1();

  }
}
```

Listing 3.1: Closure Factory returning closures which share state variable count

3.2 Local state abstraction

Closures can capture their local state of the method they are created in. E.g., the local variable *count* is accessed by both the returned closures. Due to rules of lexical scoping local state can't be referred to from outside the lexical scope. The client method can't refer to the variable *counter*, hence it can't reason about the state of variable *count*. e.g. it can't assert that after call to closure u terminates *counter* is incremented by 1.

3.3 Statically Unknown Specification

Closures can be assigned to variables, passed as formal arguments to methods or returned by methods hence the specification claims associated with them can't be statically known always. e.g. in Listing 3.1 the anonymous method assigned to variable u and d specify they require and ensure write access to *count*. A method caller should possess 100% access to *count* in order to call these methods. However method contracts of the closures are invisible to the client as it knows only the method contract of *makeCounter*. The closure creating method, also known as closure factory, should specify behavior of returned closure in its contract for the client to use.

Kassios & Muller [1] proposed a methodology to verify sequential programs in presence of closures. Each closure instance is represented by a closure object. Let S denote the type used to represent program state, C the type of closure object, T_i represent types of formal arguments and R_i represent method return types. For each closure type: $(T_1, T_2, \ldots, T_n) \rightarrow (R_1, R_2, \ldots, R_m)$ there are two pure functions *pre* and *post* declared of following types:

$$pre : (C, S, T1, T2, \ldots, Tn) \rightarrow (boolean) \tag{3.1}$$

$$post : (C, S, T1, T2, \ldots, Tn, S, R1, R2, \ldots, Rn) \rightarrow (boolean) \tag{3.2}$$

pre is a single state function which evaluates to true if precondition of the closure instance hold given the program state of its evaluation and formal arguments to the call. *post* is a two state function which asserts

that the post condition of a closure instance holds for the formal arguments, returned values and two states, the first one is state before the closure call and the second one is state after call.

Closure factory uses two fixed conditions, lets call them P and Q to specify behavior of returned closure to the client. If the factory returns a closure *foo*, it has to prove containment property as follows:

$$\forall state, ins : P \implies pre(foo, state, ins) \tag{3.3}$$

$$\forall oldstate, state, ins, outs : post(foo, oldstate, ins, state, outs) \implies Q \tag{3.4}$$

The client, in order to successfully verify a closure call has to establish that condition P holds before call and it can safely assume Q to be satisfied after call terminates. However since Chalice specifications are not side effects free, they can't be present in antecedent of logical implication. Semantics of expressions like $acc(field) \wedge field \geq 0 \implies pre(\ldots)$ are undefined in Chalice.

3.4 Shared state between closures

It is possible for two more closures to capture the same local state. In such situations, their parallel invocation would lead to race condition. E.g. in Listing 3.1, both the returned closures share access to variable *count*. If closures u and d were invoked concurrently there would be a race condition. Since *count* is hidden from client. The method *makeCounter* can't use *count* in its contract to specify u and d share the same local state. In the same example, the states captured by closures u and $u1$ are disjoint. Each call to *makeCounter* allocates different locations for *count*. Hence concurrent invocation of u and $u1$ has no race conditions. In addition to exposing abstracted values of local state another mechanism is needed for a method to abstract access permissions to local state used by closures.

Chapter 4

New Syntax

This chapter introduces new syntax constructs for using closures in Chalice by giving code snippets and describing their behavior in each section.

4.1 Closure Creation & Types

Closure Types are defined as $T_1, T_2, \cdots, T_n - - > U_1, U_2, \cdots, U_n$ where T_i and U_j denote the types of input and output parameters of the closure. Closure functions have similar notation but they are differentiated in the syntax by the use of $- >$ to separate input and output parameters. Since functions evaluate to a single expression they have single type for output: $T_1, T_2, \cdots, T_n - > U$

A new closure instance is created by assigning anonymous methods to a variable of closure type. Anonymous method definition is similar to Chalice methods except that anonymous methods don't have a name. Anonymous methods can be used only inside method bodies, currently their assignment to class fields is not supported. Closure variables are immutable hence they can be assigned only once.

```
class Counter {

  method makeCounter() returns (closure:()-->(int))
  requires /* closure specification */
  {
    closure := method(i:int)
                 requires i != 0
               { /* closure body */}
  }
}
```

Listing 4.1: Closure creation

4.2 Closure Specification

In order for a closure factory to express specifications of the closure(s) being created, two new specification operators have been added **req** and **ens**. **req** is used to specify precondition of the closure and **ens** is used to specify post conditions of the closure. In general a closure specification statement looks like :

```
k1,k2 ,..., kM = call closureVar(j1,j2 ,..., jN) req ClosurePre ens ClosurePost
```

Listing 4.2: Closure specification syntax

Variables *j1,...,jN* and *k1,...,kM* represent input and output variables of the closure *closureVar*. *ClosurePre* and *ClosurePost* represent the pre and post conditions of the closure. When a method returns a closure, it has to use a closure specification statement in its postconditions and when a method accepts closure as its formal argument, it has to use a closure specification statement in its preconditions.

req *P* is equivalent to (**req** *P* **ens** *true*) and **ens** *Q* is equivalent to (**req** *true* **ens Q**). The program in Listing 4.3 specifies that the closure *div* being created takes a variable *inp* and returns variable *out*. Call to *div* requires the input variable, *inp* to be non-zero and its termination ensures *out* to be positive if *inp* is positive.

```
method foo() returns (div:(int)-->(int))
requires call out := add(inp) req inp != 0 ens inp > 0 ==> out > 0
{
  /* ... */
}
```

Listing 4.3: Example of closure specification

Scope of the variables introduced in closure specification is limited to the *req* and *ens* construct in which they appear and their types are inferred by the closure's type.

Behavior of a closure function is described using a similar syntax as closures but function application syntax is used instead of method call. e.g. in Listing 4.4 the method *foo* returns a closure function *abs*. To describe its behavior variables *inp* and *out* are introduced using syntax of function application the specification is given using only the **req** operator.

```
method foo() returns (abs:(int)->(int))
requires out := abs(inp) req (inp < 0 => out == -inp) && (inp>=0 => out == inp)
{
  /* ... */
}
```

Listing 4.4: Example of closure function specification

In closure specifications all input/output parameters of a closure might not be of interest so an additional variable "_" has been introduced to for variable(s) which don't appear in closure specification. So if in Listing 4.5 the programmer was interested in describing only the precondition of div, the requires clause can be written as:

```
requires call _ := add(out) req out != 0
```

Listing 4.5: Ignore variable

4.3 Closures as method arguments

When a method receives a closure as one of its formal arguments, it has to express desired behavior of the closure in its preconditions. These preconditions create the necessary proof assertions to verify use of the closure inside the receiver method. In listing 4.6 the method *update* accepts a closure *callback* as its argument and calls it. Since call to *callback* needs write access to field *amount*, *update* has to specify write access to the field *amount* in its precondition. The client can then pass any closure which requires and ensures access

to field *amount* and it is guaranteed that the method *update* can invoke it successfully.

```
class Account {

  var amount:int;

  method update(callback:(Account) --> () )
    requires acc(amount)
    requires callback != null
    requires call callback(a) req acc(a.amount)
      ensures acc(amount)
  {
    call callback(this);
  }

  method updateaccount()
    requires acc(amount)
  {

    var debit100 : (Account) --> ()

    debit100 := method (a:Account)
                  requires acc(a.amount)
                  ensures  acc(a.amount)
                {
                  if (a.amount > 100) {
                    a.amount := a.amount - 100
                  }
                }

    call update(debit100)
  }
}
```

Listing 4.6: Closure as method argument

4.4 Closure Functions

Closure functions address the challenge of exposing abstraction of state captured by a closure. Like closure methods, closure functions can be assigned to variables, passed as formal arguments to a method and returned by a method. Instances of closure functions are created by assigning anonymous functions to variables of closure function type.

In Listing 4.7 the closure factory *makeCounter* returns a closure method *c* and a closure function *state*. The specification of *makeCounter* exposes the function *state* which abstracts the state of local variable *count*. The client gets to know following things about *c* and *state*:

- Evaluation of returned closure function *state* requires access to field *field*
- Call of *c* requires & ensures access to *field*
- Call to *c* ensures evaluation of *state* after the call is greater than that before the call

- c and f are not null

This example shows closure functions can be used to specify behavior of returned closure(s) and abstract hidden state. Closure function variables just like closure variables are also immutable. Closure functions are statically bounded to the closures which use them in their specifications.

In the program shown in Listing 4.7, if a local variable was used instead of an object field, closure functions are not sufficient for abstraction. Hence *makeCounter* can't use the variable *count* in its contract to specify c requires access to the local state and closure functions can't be used to abstract permissions. Usage of predicates can facilitate this mechanism as described in Section 4.5

```
class Counter{

var field :int

method makeCounter() returns (c:()-->(int), state:()->(int))
  ensures i := state() req acc(field)
  ensures call x:= c() req acc(field) ens acc(field) && state() > old(state())
  ensures  c != null && state != null
{

  state := function() :int
             requires acc(field)
           { field }

  c := method() returns (x:int)
         requires acc(field)
         ensures  acc(field) && state() == old(state()) + 1
       {
         field := field + 1
       }
}
}
```

Listing 4.7: Closure function

4.5 Closure Predicates

Closure functions abstract values of memory locations but they can't abstract permissions assertions on those locations. To solve this problem we introduce the following convention, When a closure factory returns closure(s) which capture local state, it can declare closure predicates in its contract and define them in its body, e.g. in Listing 4.8 method *counter* declares a predicate *valid* in its contract and defines it as *acc(count)*. Closure specification can then use the closure predicate to describe returned closures and closure function's behavior: closure *up* requires and ensures predicate *valid* in its pre and post conditions.

Definition of closures can use these predicates as normal chalice predicates to fold and unfold permissions on local state. Client knows only predicate declaration, it is not allowed to fold or unfold closure predicates in source language. At the point of closure call, calling thread has to possesses access to the closure predicate. For the client to possess the access to closure predicate, the factory method automatically folds the predicates declared in its contract and gives it to the calling thread at its termination.

In Listing 4.8 the method *client* automatically gets access to the predicate *valid* after call to the factory method *counter* terminates. Call of the closure variable *clos* verifies because invoking thread has access to

the predicate. If the client was modified to invoke two threads of the returned closure in parallel it would not verify. In Listing 4.9 the second statement doesn't verify because access to the predicate is given away to the new thread in the first statement.

```
method counter() returns (up: ()-->(int), iseven: ()->(bool))
  ensures call _ := up() req valid ens valid && iseven() != old(iseven())
  predicate valid
{
  var count:int;

  predicate valid { acc(count) }

  up := method() returns (x:int)
        requires valid
        ensures  valid
        {
          unfold valid
          count = count + 1;
          x := count;
          fold valid
        }

  iseven := function() : bool
            //evaluation of this function requires thread to have
            //access to predicate valid
            requires valid
            {
              unfold valid in count % 2 == 0;
            }

}

method client()
{
  val ret:int;
  var clos : ()-->(int)
  var absfunc : ()->(bool)
  var state1 :bool;
  var state2 :bool;

  call clos,absfunc := counter()
  // access to the variable count is folded by the closure factory counter
  // current thread has access to predicate valid

  // access to valid is required for function evaluation
  state1 := absfunc()

  // access to valid is given to the closure call which unfolds it
  call ret := clos()
  // access to valid is given back by the closure

  // access to valid is required for function evaluation
  state2 := absfunc()
  assert(state1 == !state2)
}
```

Listing 4.8: Closure predicates

```
fork clos()
fork clos() //error: predicate is given away in the first call
```

Listing 4.9: Closure Predicate for parallel invocation

Closure predicates are allowed to refer only memory locations local to the method in which they are defined. The reason for this is explained in Section 6.9

Chapter 5

Examples

The method *increment* of class *Math* in Listing 5.1 the method *increment* accepts a callback *adder* and requires that *adder* returns a value greater than input provided to it. Using the specifications of *adder*, *increment* can safely assert that when it calls *adder* using *inp*, its own result *res* is greater than *inp*. This example shows how methods can use specification of callbacks for their own specifications. The method *decrement* uses the same specifications to state it returns a lesser output.

```
class Math {

method increment(inp: int, adder:(int) --> (int) ) returns (res:int)
  requires adder != null
  requires call x := adder(y) ens x > y
  ensures res > inp
{
  call res := adder(inp)
}

method decrement(inp: int, adder:(int) --> (int) ) returns (res:int)
  requires adder != null
  requires call a := adder(b) ens a > b
  ensures res <= inp
{
  var temp : int;
  call temp := adder(inp)
  res := -1 * temp
}

method add10()
{
  var add10 : (int) --> (int)

  add10 := method(inp:int) returns (out:int)
             ensures out > inp
           {
             out := inp + 10;
           }

  var returnedval:int
  call returnedval := increment(5,add10)
}
```

```
}
```

Listing 5.1: Closures for callback

The method *getCounters* in Listing 5.2 returns two closures and a closure function all three of which operate on the same location referred by local variable *count*. The state of *counter* is exposed to the client using the closure function *state*. Even though the closures actually increment or decrement the value of *count* by 1, the client of method *getCounters* knows just the fact that *up* increments the state and *down* decrements it. Both the closures use same closure predicate *valid* which indicates they might be sharing the same local state, hence client can't invoke them in parallel. Using closure predicates and closure functions together, methods can abstract their states and hide implementation details from a client.

```
class Counters
{

 method getCounters() returns (up:()-->(int), down:()-->(int), state:()->(int))
   predicate valid
   ensures call x := up() req valid ens valid && ( state() > old(state()) )
   ensures call y := down() req valid ens valid && ( state() < old(state()) )
   ensures z := state() req valid
   ensures up != null && down !=null && state != null
 {
   var count:int;

   predicate valid { acc(count) }

   state := function() : int
              requires valid
            { unfolding valid in count }


   up := method() returns (x:int)
           requires valid
           ensures  valid && ( state() == old(state()) + 1 )
         {
           unfold valid
           count := count + 1
           fold valid
         }

   down := method() returns (y:int)
             requires valid
             ensures  valid && ( state() == old(state()) - 1 )
           {
             unfold valid
             count := count - 1
             fold valid
           }
 }

 method client()
 {
   var u : ()-->(int)
   var u1 : ()-->(int)
   var d : ()-->(int)
```

```
    var d1: ()-->(int)
    var f:()->(int)
    var g:()->(int)
    var i : int
    var j : int
    var c : int

    call u,d,f := getCounters()

    fork u()

    fork d()  //error, access to valid has been given away by u

    call u1,d1,g := getCounters()

    fork u1() //ok, u1 and u capture different states

  }

}
```

Listing 5.2: Com

Chapter 6

Technical Treatment

This chapter describes the formalism for verification of concurrent program with closures. Sections 6.1, 6.2, 6.3, 6.4 describe the newly introduced formalisms. Sections 6.5, 6.6, 6.7, 6.8, 6.9 describe how new syntax is verified using these formalisms.

6.1 Specification Functions

Specification functions are pure function which evaluate to boolean. Each closure is associated with two such specification functions: *pre* and *post*. They evaluate to true when the corresponding precondition/postconditions hold in the given states. As described in Section 3.3 *pre* and *post* are of the following types:

$$\begin{aligned} pre &: (C, H, M, T_1, \ldots, Tn) \rightarrow (boolean) \\ post &: (C, H, M, T_1, \ldots, Tn, H, M, R1, \ldots, Rn) \rightarrow (boolean) \end{aligned} \tag{6.1}$$

Here C represents the type to represent closure instances. H and M represent types use for Heap and Mask. T_i represent types of the input arguments and R_k represent types of the returned values.

For each closure function type, two Boogie functions are created of the following type:

$$\begin{aligned} abspre &: (C, H, M, T_1, ..., T_n, R) \rightarrow (boolean) \\ abseval &: (T_1, ..., T_n) \rightarrow (R) \end{aligned} \tag{6.2}$$

The function *abspre* holds if the preconditions of the anonymous method holds in the program state of its evaluation. The function *abseval* represents the result of evaluation of the closure instance.

6.2 Additive Translation

Let E be an expression which can have access permissions as well as logical operators. Additive transformation results in a conjunction of two boolean expressions:

$$ATr(E) \equiv AHeapTr(E) \wedge AMaskTr(E) \tag{6.3}$$

AHeapTr and *AMaskTr* are additive heap translation and additive mask translations respectively. $ATr(E)$ holds if all the heap related conditions in E hold and if all the locations present in E have at least sum of all the permissions present in E.

6.2.1 Additive Heap Translation

$AHeapTr(E)$ translates the expression E to contain only heap related assertions inside E. All permissions related expressions are translated as $true$. Following is a recursive definition of $AHeapTr(E)$.

$$
\begin{aligned}
&AHeapTr(P \Rightarrow Q) \equiv AHeapTr(P) \Rightarrow AHeapTr(Q)\\
&AHeapTr(P \wedge Q) \equiv AHeapTr(P) \wedge AHeapTr(Q)\\
&AHeapTr(acc(o.f, r)) \equiv true\\
&AHeapTr(rd(o.f, r)) \equiv true\\
&AHeapTr(E) \equiv Tr(E)
\end{aligned}
$$

$Tr(E)$ translates the source expression to BoogiePL expression as described in [3]. Tr should never encounter access permissions. The definition of ATr ensures this as Chalice access permissions acc and rd are allowed to be present only in outermost level of conjuncts and consequents of implications which are processed before Tr is used.

6.2.2 Additive Mask Translation

Additive Mask Translation expresses the minimum fractional permission to be held by each memory location present in a Chalice expression. $AMaskTr(E)$ first computes list of memory locations present in E and list of access permissions corresponding to each such location. The assertion for a particular memory location is that the location should have at least the sum of all fractional permissions associated with it.

Let us assume E contains n locations indicated by loc_i. Each of the memory locations loc_i is associated with m_i number of fractional permissions represented by $p_{i,j}$ then:

$$AMaskTr(E) = \bigwedge_{i=1}^{n} Mask[loc_i] \geq \sum_{k=1}^{m_i} p_{i,j} \tag{6.4}$$

Table 6.1 shows the computation of memory location and list of permissions associated with few Chalice expressions and Table 6.2 shows their additive mask translations.

Expression E	collectPermissions(E)
$acc(x)$	$(x, ((true, 100, 0)))$
$acc(x) \wedge x > 0$	$(x, ((true, 100, 0)))$
$acc(x) \wedge acc(x)$	$(x, ((true, 100, 0), (true, 100, 0)))$
$acc(x, 5) \wedge rd(x, 10)$	$(x, ((true, 5, 0), (true, 0, 10)))$
$acc(x, 50) \wedge rd(y, 40)$	$(x, ((true, 50, 0))), (y, ((true, 0, 40)))$
$b > 0 \Rightarrow acc(x, 50) \wedge c < 0 \Rightarrow rd(x, 40)$	$(x, ((b > 0, 5, 0), (c < 0, 0, 40)))$

Table 6.1: List of memory locations and their permissions inside a Chalice expression

In Chalice access permissions are allowed to be present in consequent of implication, hence the antecedent of the implication should be preserved together with the permission fraction.

Expression E	AMaskTr(E)
$acc(x)$	$M[x] \geq 100$
$acc(x) \wedge x > 0$	$M[x] \geq 100$
$acc(x) \wedge acc(x)$	$M[x] \geq 100 + 100$
$acc(x, 5) \wedge rd(x, 10)$	$M[x] \geq 5 + 40 \cdot \epsilon$
$acc(x, 50) \wedge rd(y, 40)$	$(M[x] \geq 50) \wedge (M[y] \geq 40 \cdot \epsilon)$
$b > 0 \Rightarrow acc(x, 50) \wedge c < 0 \Rightarrow rd(x, 40)$	$M[x] \geq ((c > 0)?50 : 0) + ((c < 0)?40 \cdot \epsilon : 0)$

Table 6.2: Additive Mask Translation

6.3 Entails Operator

Let A and B be two Chalice expressions, then we define an operator **entails**:

$$A \rightharpoonup B$$

This means expression A is stronger than expression B.

6.4 Containment

The property $Contains(P_1, Q_1, P_2, Q_2)$ holds if following assertions hold:

$$\begin{aligned} P_1 &\rightharpoonup P_2 \\ Q_2 &\rightharpoonup Q_1 \end{aligned} \tag{6.5}$$

This can be written in Boogie as following:

$$\begin{aligned} &Havoc \quad newHeap \\ &Havoc \quad newMask \\ &Inhale \quad (P_1, newHeap, newMask) \\ &Exhale \quad (P_2, newHeap, newMask) \\ \\ &Havoc \quad oldHeap \\ &Havoc \quad oldMask \\ &Havoc \quad newHeap \\ &Havoc \quad newMask \\ &Inhale \quad (Q_2, oldHeap, oldMask, newHeap, newMask) \\ &Exhale \quad (Q_1, oldHeap, oldMask, newHeap, newMask) \end{aligned} \tag{6.6}$$

The Boogie statement *Havoc* V ensures that the variable V gets an arbitrary value. Since Inhale and Exhale procedures in 6.6 use arbitrary heaps and masks, it guarantees the containment property in listing 6.5 holds for all possible heaps and masks. This methodology replaces the need for universal quantification over all program states proposed by [1]. We call it implicit state quantification.

6.5 Closure as Return Parameters and Method Arguments

One of the proof obligations of a method returning a closure is that the specification of the returned closure should be "contained" in the specifications of the closure as ensured by the factory. Similarly when a closure is passed as an argument to a receiver method, the caller has to establish that the specification of closure being passed are "contained" with specifications of the closure as required by the receiver method. The actual specifications of a closure have to be contained in the exposed specifications.

Let the exposed pre/post conditions be (P_1, Q_1) and actual pre/post conditions be (P_2, Q_2) then verification of the method passing or returning a closure has to establish the containment property mentioned in Listing 6.5.

When a method returning a closure has to establish containment property, P_1 are Q_1 are the specifications of the closure in its postcondition. When a method caller passes a closure as an argument, P_1 are Q_1 are the specifications of the closure exposed by the receiver method's preconditions. P_2 and Q_2 are the specification functions obtained by additive translation at the time of closure creation as mentioned in Section6.1.

$$\begin{aligned}&(P_1 \rightharpoonup pre(c, newHeap, newMask, i_1, ..., i_n)) \\ &(post(c, oldHeap, newMask, i_1, ..., i_n, newHeap, newMask, o_1, ..., o_m) \rightharpoonup Q_1)\end{aligned} \tag{6.7}$$

Since closure specifications are statically bound to a closure variable 6.7 can be simplified by using the exact specifications instead of specification functions.

6.6 Closure Creation

When a closure instance is created two assertions are generated regarding pre and post conditions for the instance of newly created closure.

$$\forall h, m, ... :: pre(c, h, m, ...) \equiv ATr(Cpre) \tag{6.8}$$

$$\forall oh, h, m, ... :: post(c, oh, ..., h, m, ...) \equiv ATr(Cpost) \tag{6.9}$$

Where ATr is additive translation of the specification for the closure and $CPre$ and $CPost$ are the fixed pre and post conditions of closure specified by using **req** and **ens** operators.

6.7 Closure Call & Concurrent Invocation

Usage of specification functions *pre* and *post* are not sufficient to verify closures call or their concurrent invocation. They can either give assertion on the program state before method call or they can give assumptions on the program state after the method call. Specification functions can't alter the state of the program unlike Chalice Inhale and Exhale procedures. As a solution closure specifications are statically bound to the closure variable. For this methodology to work correctly, all closure variables are made immutable. This ensures fixed specifications can be used for verification of closures. E.g., in the program listed in 6.1, the call to method *foo* assigns the returned closure to the variable x and binds the specifications of method output parameter *retval* from method contracts of *foo*. The actual specifications of *retval* inside the body of *foo* might be different from those given in it's contract. Now that fixed expressions for specification of closures are known, verification steps for closure call and concurrent invocation are similar to those for Chalice methods.

```
class Counter {
  var field :int;

  method foo() returns (retval:()-->())
  ensures call retval() req acc(field) ens acc(field) && field > 0
  {
    /* ... */
  }

  method bar()
  requires acc(field)
  {
     var x:(int) --> (int)

     //x gets precondition: acc(field) and postcondition: acc(field) && field > 0
          call x := foo()

          //Inhale acc(field)
          call x()
          //Exhale acc(field)

          //Inhale acc(field)
          fork x()

          //error location not writable
          field := 10
  }

}
```

Listing 6.1: Parallel invocation of closure

6.8 Closure Functions

6.8.1 Creation

Each closure function is always associated with an instance. Closure function creation establishes following assertions:

$$\begin{aligned} &\forall h, m, i_1, \ldots, i_n :: abspre(f_{obj}, h, m, i_1, \ldots, i_n) \equiv ATr(Func_{pre}) \\ &\forall h, m, i_1, \ldots, i_n :: abseval(f_{obj}, h, m, i_1, \ldots, i_n) \equiv Tr(Func_{def})) \end{aligned} \tag{6.10}$$

Where $i_1, \ldots, i_n$ represent the input arguments of the function. f_{obj} represents the newly created function instance, $Func_{pre}$ represents the precondition of the function, $Func_{def}$ represents its definition. ATr and Tr are additive translation and Chalice translation respectively as described in 6.2

6.8.2 Containment

Verification of closure function creation is verification of containment property, with respect to the preconditions:

$$P \rightharpoonup pre(c, newHeap, newMask, i1, ..., in)$$

Where P is precondition of the specification function given by the factory method.

6.8.3 Evaluation

Verification of closure function evaluation requires evaluating thread to possess the permissions required in preconditions of the function. Since closure functions like static Chalice functions are side effect free, they don't Inhale or Exhale when they are evaluated. So a function application statement like:

```
v := closfunc(10)
```

gets translated to following statements:

$$v := abseval(closfunc, H, M, 10)$$

6.9 Closure Predicates

Closure predicates act as normal Chalice predicates for anonymous method definitions inside the method creating them. When a method declaring a closure predicate is called, the verification framework automatically folds the closure predicate at method termination. That is, all access permissions present in the predicate definition are taken away and 100% permission of the predicate is given to the closure object(s) being returned. When a closure object is invoked, it should posses access of the closure predicate instance. Since all the closures returned by same method call share the same local environment, they also share the same predicate instance. Hence this methodology ensures their is no race condition on access to the shared local state between closures. Closure predicate generated by different calls to the same method will generate distinct instances of closure predicates.

Closure predicates should abstract only the memory locations of local variablss because two or more closures can always refer to the same "non-local" environment. E.g., in Listing 6.2 if we don't limit the predicate *valid* to local variables it can refer to field *field*. Forking *c1* twice is correctly caught as a race condition because they share the same environment. But forking of *c2* is verified incorrectly even though there is a race condition on location of *field*. Since predicate definitions are not available to the client the information that *c1* and *c2* access shared state is hidden.

```
class A {
  var field : int;

  method factory() returns (retval:()-->())
  ensures call retval() req valid ens valid
  predicate valid
  {
    predicate valid { acc(local) && acc(field) }

    var local: int;

    retval := method()
              requires valid
              ensures valid
```

```
            {
              unfold valid
              //... local and field can be modified
              fold valid
            }
}

method client()
{
   var c1:() --> ()
   var c2:() --> ()

         call c1 := factory()
         //a new folded predicate p1 assigned to c1

         call c2 := factory()
         //a new folded predicate p2 assigned to c2

         fork c1;

         fork c1; //error correctly caught, c1 has already given away access to p1

         fork c2; //incorrectly verifies, c1 still has access to field

   //incorrect, as c1 and c2 can update the field variable concurrently
}

}
```

Listing 6.2: Closure predicates to abstract access local state

Chapter 7

Conclusions

Related Work

Kassios and Mueller[1] have proposed a methodology for the specification and verification of closures for sequential programs in first order logic. They used universal quantification over all program states to verify that a closure's actual specifications are contained inside its exposed specifications. We use implicit state quantification to solve the problem without using first order formalization.

A significant difference between verification methodologies of [1] and Chalice is the way the framing problem is tackled. Both the methodologies use modular verification, that is, they use a method's contract and not its implementation details to verify the method's use. This requires the method contracts to contain frames, which give an upper bound on the set of memory locations modifiable by the method. In [1], the authors use Dynamic Frames[2] where method contracts should contain a **mod** clause in the source language to specify the method's frame. Chalice uses Implicit Dynamic Frames [5][3] where the frame of a method can be derived from access permissions present inside its specifications. We have used Implicit Dynamic Frames by statically binding specifications to closure variables. This approach required us to limit closure variables to be immutable.

Our state abstraction mechanism uses ideas from both Chalice and [1]. Closure functions, like Chalice pure functions, specify preconditions that a thread should possess in order to evaluate them. The verification of closure functions uses abstraction specification functions as defined in [1]. To abstract over permissions on local state we have used Chalice predicates to define closure predicates.

Issues

In the Boogie statements generated for verification of Chalice programs, havoc statements are used on the current heap of the program, which assign an arbitrary value to the heap. The statements after such havoc statements work on the assumption that the state of local variables have not been modified as they are allocated on program stack. In our methodology, since we have allocated local variables on heap, their values are lost after current heap is "havoced". This issue causes test cases given in the existing Chalice repository to fail. A fix of this problem can be to restore values of the local variables in the newly havoced heap.

Due to time restrictions, the rule that a closure predicate should use only local variables in its definition has not been implemented in the program verifier. This restriction should be implemented in the type checking phase.

Extensions

We have used immutable variables to represent closure instances. If we allow closure variables to be mutable, their specifications can't be attached statically. To solve this problem, a possible future direction of work is to use specification functions from [1] together with Implicit Dynamic Frames.

For the sake of simplicity, closures were restricted to be used as local variables, method arguments or method return values. As a future work, it would be useful to study interaction of closure with other language features e.g. fields and channels.

Concluding Remarks

In this thesis we have proposed a verification methodology for concurrent programs with closures. We solved the framing problem using Implicit Dynamic Frames. We gave an extension to the Chalice language by introducing new syntax elements and showed how challenges identified are addressed by the proposed methodology. All the examples shown in the document have been verified using the extended Chalice verifier. We have created a comprehensive suite of test cases (not included in report) to cover different aspects of the newly introduced language extension.

Bibliography

[1] Ioannis T. Kassios & Peter Mueller, Specification and Verification of Delegates in First Order Logic.

[2] Ioannis T. Kassios, Dynamic frames: Support for framing, dependencies and sharing without restrictions, FM06, volume 4085 of Lecture Notes In Computer Science, pages 378-393. Springer-Verlag, 2009

[3] Leino and Mueller, A basis for verifying multi-threaded programs. Programming Languages and Systems, volume 5502 of Lecture Notes In Computer Science, pages 268-283. Springer-Verlag, 2006

[4] k. R. Leino, Peter Mueller and Jan Smans, Verification of Concurrent Programs with Chalice. In Foundation of Security Analysis and Design, Lecture Notes in Computer Science, page 195-222. Springer, 2009.

[5] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames. In FTfJP 2008, pages 1-12,2008. Technical Report ICIS-R08013, Radboud University

[6] J. Boyland. Checking interference with fractional permissions. In SAS 2003, volume 2694 of Lecture Notes in Computer Science, pages 55-72. Springer, 2003

[7] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In FMCO 2005, volume 4111 of Lecture Notes in Computer Science, pages 364-387. Springer, 2006

[8] Leonardo de Moura, Nikolaj Bjørner Z3: An efficient SMT solver. In TACAS 2008, volume 4963 of Lecture Notes in Computer Science, pages 337-340. Springer, 2008.

[9] ECMA. C# language specification. Technical Report Standard 334, ECMA, 2006.

[10] M. Odersky, L. Spoon, and B. Veners. Programming in Scala. Artima, 2007

[11] D. M. Beazly. Python Essential Reference. SAMS, 3rd edition, 2006.

[12] D. Flanagan, Y. Matsumoto. The Ruby Programming Language. OReilly, 200