Translating Chalice into SIL

Report

Christian Klauser
klauserc@student.ethz.ch

October 22, 2012

- Use either `acc(x.f,write)` (SIL toString) or `acc(x.f,full)` (SIL API), not both!
- Don't use abbreviated read fraction syntax `rd(x.f)`, ever.

1. Introduction

1.1 Semper Project

2. Background

2.1 Chalice

Chalice is a research programming language with the goal of helping programmers detect bugs in their concurrent programs. As with most languages aimed at automatic static verification (e.g, Spec#), the programmer provides annotations that specify how they intend the program to behave. These annotations appear in the form of monitor invariants, loop invariants and method pre- and postconditions. A verification tool can take such a Chalice program and check statically that it never violates any of the conditions established by the programmer.

The original implementation of the automatic static program verifier for Chalice generates a program in the intermediate verification language Boogie [**?**]. A second tool, conveniently also called Boogie, takes this intermediate code and generates verification conditions to be solved by an SMT solver, such as Z3 [**?**].

Listing 1 demonstrates how we can implement integer division and have the verifier ensure that our implementation is correct. Our solution repeatedly subtracts the denominator `b` until the rest `r` becomes smaller than `b`. Because this exact algorithm only works for positive numerators and denominators, the method **`requires`** that the numerator `a` is not negative and that the denominator is strictly positive.

Similarly, we specify what the method is supposed to do: the **`ensures`** clause tells the verifier that, when our method is ready to return, the resulting quotient `c` must be the largest

Listing 1: Loop invariants, pre- and post conditions in a Chalice program

```
class Program {
  method intDiv(a : int, b : int) returns (c : int)
    requires 0 <= a && 0 < b;
    ensures c*b <= a && a < (c+1)*b;
  {
    c := 0;
    var r : int := a;
    while(b <= r)
      invariant 0 <= r && r == (a - c*b)
    {
      r := r - b;
      c := c + 1;
    }
  }
}
```

Listing 2: Chalice example of object creation and (write) accessibility predicates.

```
class Cell { var f : int }
class Program {
  method clone(c : Cell) returns (d : Cell)
    requires c != null && acc(c.f)
    ensures acc(c.f)
    ensures d != null && acc(d.f) && d.f == c.f
  {
    d := new Cell;
    d.f := c.f;
  }
}
```

integer for which $c \cdot b \leq a$ still holds. If the verifier cannot show that this postcondition holds for all invocations of this method that satisfy the precondition, it will reject the program.

The final bit of annotation in this example is the **invariant** on the **while** loop. A loop invariant is a predicate that needs to hold immediately before the loop is entered and after every iteration, including the last one where the loop condition is already false. This annotation helps the verifier understand the effects of the loop without knowing how many iterations of the loop would happen at runtime.

2.1.1 Permissions

What sets Chalice apart from other languages for program verification is its handling of concurrent access to heap locations. Whenever a thread wants to read from or write to a heap location it requires read or write permissions to that location, respectively. These permissions only exist for verification and would be erased by compilers for Chalice.

Listing 3: Calling `Program::clone` (extension of Listing 2)

```
1  class Program {
2    //...
3    method main()
4    {
5      var c : Cell :=  new Cell;
6      c.f := 5;
7      var d : Cell;
8      call d := clone(c);
9      assert d.f == 5; // will fail, c.f might have changed
10   }
11 }
```

As an under-approximation of the set of permissions a thread would have at runtime, Chalice tracks permissions for each method invocation (stack frame, activation record). That way, the verifier can verify method bodies in complete isolation of one another. The programmer thus has to specify which heap locations need to be accessible for each method.

In Listing 2, we use *accessibility predicates* of the form `acc(receiver.field)` in the method's pre- and postcondition. `acc(c.f)` in the precondition allows us to refer to `c.f` in the method body. The accessibility predicates in the postcondition, on the other hand, represent permissions that the method will have to "return" to its caller upon completion. Conceptually, the caller passes the permission requested by the callee's precondition on to the callee. Similarly, the caller receives the permissions mentioned in the callee's postcondition when the call returns. As a consequence of verifying each method in isolation, it doesn't matter whether a method is called on the same thread or on a thread of it's own (with the caller waiting for the callee's computation to finish). The necessary permissions need to be transferred in both scenarios.

Listing 3 demonstrates how our `clone` method could be used. Unfortunately, the assertion on line 9 will fail, as the verifier has to assume that `clone` might have changed the value stored in `c.f`. In Chalice, whenever a method gives away all permissions to a memory location (so that it doesn't even have read-access), it must assume that that location has been changed, the next time it gets to read said location. While we might change augment the postcondition of `clone` with the requirement that `c.f == old(c.f)` (the value of `c.f` at method return must be the same as it was on method entry), there is a much more elegant solution to this problem: *read-only permissions*.

2.1.2 Percentage Permissions

When Chalice was originally created, the programmer could specify read-only permissions as *integer percentages* of the full (write) permission. `acc(x.f,100)` is the same as `acc(x.f)`, i.e. grant read and write access, whereas any other strictly positive percentage `acc(x.f,n)` for only grants read access to the heap location `x.f` ($n \in \mathbb{N}, 0 < n < 100$). The verifier keeps track of the exact amount of permission a method holds to each heap location, so that write-access is restored when a method manages to get 100% of the permission back together, after having handed out parts of it to other methods or threads.

While percentage permissions are very easy to understand, they have the serious draw-

back that the number of percentage points of permission a method receives to a certain location, essentially determine the maximum number of threads with (shared) read access that method could spawn. That is a violation of the procedural abstraction that methods are intended to provide.

2.1.3 Counting Permissions

Another drawback of percentage permissions is, that it is difficult to deal with a dynamic number of threads to distribute read access over. As a solution to that problem, Chalice also introduced *"counting permissions"* that are not limited to just 100 "pieces" of permission. Accessibility predicates using counting permissions are written as `acc(x.f,rd(1))` and denote an arbitrarily small but still positive (non-zero) amount of permission ε. Permission amounts equal to multiples of ε can be written as `acc(x.f,rd(n))`, but any finite number of epsilon permissions are defined to be still smaller than 1% of permission. This also means that a method that holds at least 1% of permission, can always call a method that only requires $n \cdot \varepsilon$ of permission.

Unfortunately, counting permissions (often also referred to as *"epsilon permissions"*) still cause method specifications to leak implementation details. An epsilon permission cannot be split up further, thus a method that acquires, say, 2ε of permission to a heap location cannot spawn more than two threads with read access to that heap location.

2.1.4 Fractional (Read) Permissions

In order to regain procedural abstraction [**?**] added an entirely new kind of permission to Chalice: the fractional read permission, based on [**?**]. The idea is to allow for "rational" fractions of permission because, unlike epsilon or percentage permissions, those could always be divided further. Composability can still be an issue, even with rational permissions. A method that requires $\frac{1}{107}$ of permission could still not be called from a method that only has $\frac{1}{137}$, even though the fractions passed around the entire system could almost alway be re-scaled to make that call possible. Thus, instead of forcing the programmer to choose a fixed amount of permission ahead of time, all accessibility predicates involving fractional permissions are kept *abstract*.

The programmer writes `acc(x.f,rd)` to denote an abstract (read-only) accessibility predicate to the heap location `x.f`. The amount of permission denoted by `rd` is not fixed. When used in a method specification, the `rd` can represent a different amount of permission for each method invocation.

To make fractional permissions actually useful, Chalice applies certain constraints to the amount of permission involved in `acc(x.f,rd)`. Firstly, fractional read permissions always represent a fraction of the caller's permission. When a caller gives away a fractional read permission to a heap location, it will always retain read-access to that location. That way, the caller can be sure that the contents of the memory location don't change. Secondly, a common idiom in Chalice is to have methods that return the exact same permissions they acquired in the precondition back to the caller via the postcondition. When a method requires `acc(x.f,rd)` and then ensures `acc(x.f,rd)`, we would want these two amounts of permission to be the same. That way, a caller that started out with write access to `x.f` gets back the exact amount of permission it gave to our method.

Listing 4: Corrected example using abstract read permissions

```
1  class Cell { var f : int }
2  class Program {
3    method clone(c : Cell) returns (d : Cell)
4      requires c != null && acc(c.f,rd)
5      ensures acc(c.f,rd)
6      ensures d != null && acc(d.f) && d.f == c.f
7    {
8      d := new Cell;
9      d.f := c.f;
10   }
11
12   method main()
13   {
14     var c : Cell :=  new Cell;
15     c.f := 5;
16     var d : Cell;
17     call d := clone(c);
18     assert d.f == 5; // will now succeed
19     c.f := 7; // we still have write access
20   }
21 }
```

Chalice restricts read fractions in method specifications even further: all fractional read permissions in a method contract, even to different heap locations, refer to the same amount of permission (but that amount can still differ between method invocations). This restriction accounts for the limited information about aliasing available statically and also makes the implementation of fractional read permissions more straightforward.

Listing 4 shows the corrected version of our example above (Listings 2 and 3) using (abstract) read permissions (`acc(c.f,rd)` in lines 4 and 5). Note that we don't need to tell the verifier that `c.f` won't change separately, because it uses permissions to determine what locations can be modifed by the method call to `clone`.

2.1.5 Fork-Join

As a language devoted to encoding concurrent programs, Chalice has a built-in mechanism for creating new threads and waiting for threads to complete in the familiar *fork-join* model. Replacing the **`call`** keyword in a (synchronous) method call with **`fork`** causes that method to be executed in a newly spawned thread. As with a synchronous method call, the caller must satisfy the callee's precondition and will give all permissions mentioned in that precondition.

```
fork tok := x.m(argument1, argument2, ..., argumentn);
// do something else
join result1, result2, ..., resultn := tok;
```

Listing 5: Alternative definition of `Cell` using functions.

```
class Cell {
  var f : int
  function equals(o : Cell) : bool
    requires acc(f,rd)
    requires o != null ==> acc(o.f,rd)
  { o != null && f == o.f }
}
```

While just forking off threads might work for some scenarios, most of the time the caller will want to collect the results computed by its worker threads at some point. To that end, the **fork** statement will return a *token* that the programmer can use to have the calling method wait for the thread associated with the token to complete. The permissions mentioned in the postcondition of the method used to spawn off the worker thread will also be transferred back to the caller at that point.

2.1.6 Information Hiding through functions and predicates

A major shortcoming of pre- and postconditions as presented so far, is that they often "leak" implementation details. One example of this happening is the `clone` method from listing 4. It ensures that the values from the old object are copied over to the newly created object, but in the process tells the caller that there is exactly one field, called `f` on those objects. Should the definition of class `Cell` ever change, sifting through the entire program and updating specifications is going to be in order. What the programmer wanted to say is, that the two objects are "*equal*".

Functions help cut down code repetition and put an abstraction layer between the implementation of a method and its clients. Listing 5 presents an alternative definition of `Cell` that exposes the equality testing function `equals`. Below is a corresponding signature for the method `clone` that uses this function. If we were to add a new field to `Cell` now, callers of `clone` would no longer see a change in the method's signature.

```
method clone(c : Cell) returns (d : Cell)
    requires c != null && acc(c.f,rd)
    ensures acc(c.f,rd)
    ensures d != null && acc(d.f) && c.equals(d)
```

Notice how the `equals` function does *not* have a postcondition that describes the function's result or "returns" permissions back to the caller. This is because functions are little more than abbreviations of common expressions. In order to be used in pre- and postconditions, they are forbidden from changing any state, which is why the programmer doesn't have explicitly return permissions to the function's caller. This happens automatically.

Predicates, on the other hand, are a way to abstract over not just values but also over accessibility. Additionally, unlike functions, they are treated as abstract entities unless the programmer explicitly "unfolds" them to apply their definition. When a method requires a predicate in its precondition, it will not automatically get the permissions (and other assertions) "contained" in the predicate because at that point, the predicate acts like a black box. The method can pass the predicate to other methods or threads and it behaves

Listing 6: Using the predicate `valid` to hide the representation of `Indentation`

```
class Indentation {
    var count : int;

    predicate valid
    { acc(count) && 0 <= count }

    function getCount() : int
        requires valid;
    { unfolding valid in count }

    method increase(amount : int)
        requires valid && 0 <= amount;
        ensures valid;
        ensures old(getCount()) + amount == getCount();
    {
        unfold valid;
        count := count + amount;
        fold valid;
    }
}
```

much like a permission to a memory location: it cannot be duplicated and once given away, it's gone.

Given a predicate, the programmer can use the **`unfold`** statement to "trade" the predicate for its definition. The current thread will receive all permissions "contained" in the predicate and gets to assume any other assertions associated with the predicate. After the programmer is done operating on the predicate's contents, they can use **`fold`** to "trade" access permissions in exchange for the predicate.

Listing 6 additionally demonstrates the **`unfolding`** expression syntax used to temporarily getting access to the contents of a predicate.

2.1.7 Monitors (locks)

Using just fork-join, it is impossible for threads to communicate with one another. They can only produce a result and all of their memory writes only become visible when they return the exclusive write permissions back to their caller. To handle more realistic scenarios, such as concurrent access to a shared queue, Chalice comes with *monitors* that allow for exclusive locking of a shared resource. For each class, the programmer can define a *monitor invariant* that represents the "resources" that the monitor is supposed to manage access to. As with predicates, this definition can consist of both accessibility predicates and ordinary boolean assertions.

Initially, objects are not available for locking via the monitor mechanism. When the programmer *shares* an object with other threads using the **`share`** statement, the access permissions associated with the invariant get stored in the monitor (similar to **`fold`** for predicates). Threads that subsequently **`acquire`** the lock on this *shared* object will receive the

Listing 7: Example of the life-cycle an object can go through in Chalice

```
class C {
    var f : int;

    invariant acc(f);

    method main(){
        var c : C := new C;
        c.f := 5;
        share c;
        acquire c; c.f := 7; release c;
        // cannot access c.f here
        acquire c; c.f := 6; unshare c;
        assert c.f == 6;
    }
}
```

contents of the monitor (similar to **unfold**). The object is now *locked* and can be made available to other threads via the **release** statement (similar to **fold**, again). The programmer can also revert the conversion to a *shared* object by using the **unshare** statement (similar to **unfold**, again). Listing 7 demonstrates these statements with a single thread.

As with monitors in Java and C#, in order to guarantee mutual exclusion, threads that reach an **acquire** statement are blocked until the monitor can grant them the exclusive lock. With such a simple blocking mechanism comes the risk of deadlocks (thread 1 waiting for monitor *b*, currently held by thread 2, which is waiting for monitor *a*, currently held by thread 1).

To solve this problem, the Chalice verifier makes sure that locks are acquired in a fixed order. The programmer can assign a *locking level* to a monitor, ensuring that the lock on that monitor can only be acquired when that locking level is *higher* than the locking level of all other locks held by the current thread. Whether one locking level is higher than another, is determined by a strict partial order that we denote as $<<$. The **share** statement seen above optionally accepts clauses of the form **between** ... and ..., **above** ... or **below** ... to constrain the *lock level* at which the monitor is installed. If such a clause is missing, Chalice chooses `above waitlevel`, which means that the lock level is higher than the highest lock level of all locks currently held by the thread (we refer to this maximum as a thread's *wait level*).

In listing **??**, we create two objects `a` and `b` and share them. The lock level of `a` defaults to `above waitlevel` and the programmer explicitly declare the lock level of `b` to be `above a`. This means that if a thread plans to lock both `a` and `b`, it will have to first lock on `a` then `b`. Should the programmer try to lock objects in the opposite order, on **acquire** `a` the thread's wait level would already be at the lock level of `b`, which is above `a`'s.

Lock levels are implemented via a special field called `mu` of type `Mu` (the type of lock levels), available on every object. The `mu` field is assigned during **share** and **unshare** operations and needs to be read-able for acquiring the lock.