Translating Chalice into SIL

Report

Christian Klauser
`klauserc@student.ethz.ch`

August 13, 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. Background

1.1 Chalice

Chalice is a research programming language with the goal of helping programmers detect bugs in their concurrent programs. As with Spec#, the programmer provides annotations that specify how they intend the program to behave. These annotations appear in the form of object 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 [BDJ+06]. A second tool, conveniently also called Boogie, takes this intermediate code and generates verification conditions to be solved by an SMT solver. In its default configuration, Boogie forwards its output to Z3 [dMB08].

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 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

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;
  }
}
```

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.

1.1.1 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.

As an over-approximation of the set of permissions a thread will 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

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 }
```

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. This is a consequence of treating methods as if they would each be executed in their own thread. 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.

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`. While we might change augment the postcondition of `clone` with the requirement that `c.f == old(c.f)` (the vaue 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*.

1.1.2 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)` would be the same as `acc(x.f)`, i.e. grant write access, whereas `acc(x.f,50)` would only grant read access to the heap location `x.f`. The verifier would keep 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% permission amount 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 drawback 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 inteded to provide.

1.1.3 Counting Permissions

Another drawback of percentage permissions is, that it is difficult to deal with a dynamic number of thread to distribute read access over. Accessibility predicates using counting permissions are written as `rd(x.f,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 rd(x.f,n), but by definition any finite number of epsilon permissions is still smaller than 1% of permission. This also means that a method that holds at least 1% of permission, can alway 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.

1.1.4 Fractional (Read) Permissions

In order to regain procedural abstraction [HRL+11] added an entirely new kind of permission to Chalice: the fractional read permission. The idea is to allow for "rational" fractions of permission because, unlike epsilon or percentage permissions, those could always be divided further. Composability would 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 static. 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). A common idiom in Chalice are methods that return the exact same premissions they acquired in the precondition back to the caller via the postcondition. When a method requires acc(x.f,rd) and then ensures (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.

Chalice restricts read fractions in method specifications even further: all fractional read permissions in a method contract 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.

1.1.5 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

Listing 4: Corrected example using abstract read permissions

```
class Cell { var f : int }
class Program {
  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) && d.f == c.f
  {
    d := new Cell;
    d.f := c.f;
  }

  method main()
  {
    var c : Cell :=  new Cell;
    c.f := 5;
    var d : Cell;
    call d := clone(c);
    assert d.f == 5; // will now succeed
    c.f := 7; // we still have write access
  }
}
```

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;
```

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.

- Inhale, Exhale, Havoc, terms used in Boogie implementation

- Methods

- Fork+Join

- Predicates + Functions

- Monitors

References

[BDJ+06] M. Barnett, R. DeLine, B. Jacobs, B.-Y. E. Chang, K. Rustan, and M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium, volume 4111 of Lecture*. Springer, 2006.

[dMB08] L. de Moura and N. Bjørner. Z3: An efficient smt solver. In *In TACAS 2008, volume 4963 of LNCS*, pages 337–340. Springer, 2008.

[HRL+11] Stefan Heule, K. Rustan, M. Leino, P. Müller, and A. J. Summers. Fractional permissions without the fractions. *Formal Techniques for Java-like Programs (FTfJP)*, 2011.