Translating Chalice into SIL

Report

Christian Klauser
klauserc@student.ethz.ch

October 15, 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

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 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, such as 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.

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;
  }
}
```

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.

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 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 }
```

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. 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*.

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)` 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.

2.1.3 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.

2.1.4 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`.

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
  }
}
```

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;
```

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 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 }
}
```

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 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.

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;
    }
}
```

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 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).

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;
    }
}
```

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 8, 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.

2.1.8 Details on the Boogie-based Chalice verifier

Is "Inhale, exhale and havoc" a better title for this section?

In order to verify Chalice programs, the Boogie-based verifier permission transfer is modelled by two operations: *inhale* and *exhale*.

Listing 8: Example of deadlock-prevention

```
class C {
    var f : int;
    invariant acc(f);

    method main() {
        var a := new C;
        share a;
        var b := new C;
        share b above a;

        acquire a; acquire b;
        release b; release a;

        acquire b;
        acquire a; // illegal
    }
}
```

2.2 Semper Intermediate Language (SIL)

3. Translation of Chalice

High-level overview + including focus areas

3.1 Fractional Read Permissions

To SIL, permission amounts are just another data type. The SIL prelude only defines a set of constructors (such as no permission, full permission) and some operators and predicates (such as permission addition, subtraction, equality, comparison). In particular, it does not specify how permissions are represented. This aligns well with the abstract nature in which fractional permissions are written by the programmer. As with previous verification backends for Chalice, concrete permission amounts associated with fractional read permissions (`acc(x.f,rd)`) are never chosen but only constrained. This also means that two textual occurrences of `acc(x.f,rd)` in different parts generally do not represent the same amount of permission.

Not choosing a fixed permission amount for abstract read permissions makes them very flexible. As long as a thread holds any positive amount of permission to a location, we know that we can give away a smaller fraction to a second thread and thereby enable both threads to read that location. Unfortunately, that amount of flexibility would also make fractional read permissions very hard to use, since every mention of a read permission could theoretically refer to a different amount of permission. Chalice, therefore, imposes additional constraints on fractional permissions involved in method contracts, predicates, and monitors. In the following sections we will describe how Chalice2SIL handles each of these situations.

Listing 9: A call that uses and preserves fractional read permissions.

```
class Actor {
  method main(a : int) returns (r : Register)
    ensures r != null
    ensures acc(r.val)
    ensures t.val == a
  {
    r := new Register;
    r.val := 5;
    call act(r);
    r.val := a; //should still have write access here
  }

  method act(r : Register)
    requires r != null
    requires acc(r.val,rd)
    ensures acc(r.val,rd)
  { /* ... */ }
}
class Register {
  var val : int;
}
```

Listing 10: Handling of fractional read permissions by the Boogie-based Chalice verifier.

```
procedure act(r : Register)
{
  var k_m;
  assume (0 < k_m) && (k_m < Permission$FullFraction);
  // inhale (precondition), using k_m for rd
  ...
  // exhale (postcondition), using k_m for rd
}
```

3.1.1 Methods and fractional permissions

In Chalice programs, a very common pattern is that a method "borrows" permissions to a set of locations, performs its work and then returns the same amount of permission to the method's caller. In order to readily support this scenario, the original implementation of fractional permissions in Chalice constrains the various fractions mentioned in a method's pre- and postcondition to a value that is chosen once per call site.

For verifying the callee in listing 9, the Boogie-based implementation introduces a fresh variable permission variable k_m, constrains it to be a read-permission ($0 < k_m <$ full) and uses it in pre- and postconditions whenever it encounters the abstract permission amount `rd`.

Notice how the Boogie-based encoding of Chalice in listing 10 does not make use of the pre- and postcondition mechanism provided by Boogie. This is primarily because Boogie

Listing 11: Handling of fractional read permissions by the Chalice2SIL translator

```
method Actor::act(r : Register, k_m : Permission)
  requires 0 < k_m && k_m < write
  requires r != null
  requires acc(r.val, k_m)
  ensures acc(r.val, k_m)
{ … }
```

does not have a concept of inhaling and exhaling of permissions. Not so with SIL, which features pre- and postconditions that are aware of access predicates. Conceptually, when you "call" a method in SIL, the precondition is properly exhaled and the postcondition inhaled afterwards.

However, using SIL preconditions also means that we can't just make up a new variable k_m, instead it becomes a "ghost" parameter and introduces an additional precondition. This makes a lot of sense, since the value k_m is always specific to one call of a method.

In the actual Boogie-based encoding, the upper bound on `k_m` is even lower to give the programmer more flexibility. Currently, `k_m` is assumed to be smaller than a thousandth of `1%`. This allows the programmer to for instance specify a method that requires `acc(x.f,rd)` twice, effectively demanding at least `2*k_m` permission to `x.f`. The exact ratio was chosen arbitrarily and could always be lowered, but has so far worked well for most examples.

3.1.2 Method calls with fractional permissions

Without fractional permissions, synchronously calling a method in SIL is as simple as using the built-in call statement:

```
call () := Actor::act(r)
```

SIL takes care of asserting the precondition, exhaling the associated permissions, havocing the necessary heap locations, inhaling the permissions mentioned by the postcondition and finally assuming said postcondition. Adding support for fractional read permissions now only means providing a call-site specific value k, right?

Unfortunately, this where the high-level nature of SIL becomes an obstruction. For each method call-site, we want to introduce a fresh variable k_c that represents the fractional permission amount of permission selected for that particular call. Then, we want to constrain it to be smaller than the amount of permissions we hold to each of the locations mentioned with abstract read permissions (`rd`). For the simple preconditions above, this is easy to accomplish:

```
var k_c : Permission;
assume k_c < perm(r.val);
call () := Actor::act(r,k_c);
```

The term `perm(r.val)` is a native SIL term that represents the amount permission the current thread holds to a particular location. Sadly, this simple scheme breaks down when we have to deal with multiple instances of access predicates to the same location.

Chalice dictates that

```
exhale acc(x.f,rd) && acc(x.f,rd)
```

is to be treated as

```
exhale acc(x.f,rd)
exhale acc(x.f,rd)
```

Both exhale statements cause k_c to be constrained to the amount of permission held to $x.f$. Since exhale has the "side-effect" of giving away the mentioned permissions, this k_c will be constrained further by the second exhale statement.

Additionally, access predicates can be guarded by implications. In that case, the Boogie-based Chalice implementation translates

```
exhale P ==> acc(x.f, rd)
```

as

```
if(P)
{
  exhale acc(x.f, rd);
}
```

At this point we could have decided not to use SIL's built-in call statement and instead encode synchronous method calls as a series of exhale statements, followed by inhaling the callee's postcondition. While that would have been equivalent from a verification perspective, we would still be throwing away information: the original program's call graph.

In order to still use SIL's call statement, we need to keep track of the "remaining" permissions while constraining k_c without actually giving away these permissions, otherwise the verification of the call statement would fail. We cannot simply create a copy of the permission mask as a whole and have exhale operate on that instead. SIL at least allows us to look up individual entries of the permission mask via the `perm(x.f)` term. We use that ability to manually create and maintain a permission map of our own.

Like the permission mask in the Boogie-encoding of Chalice, this data structure must map heap locations, represented as pairs of an object reference and a field identifier, to permission amounts. At this time, SIL has no reified field identifiers. So in order to distinguish locations (pair of an object reference and a field), the Chalice2SIL translator assigns a unique integer number to each field in the program.

The only way to populate this map, is to "copy" the current state of the actual permission mask entry by entry via the `perm(x.f)` term. Unfortunately, we can't do this in one big "initialization" block, since some of the object reference expressions that occur on the right-hand-side of implications might not be defined outside of that implication.

We could expand implications in the precondition twice: once for initializing our permission map, and once to actually simulate the exhales and constraining of k_c, but there is a more concise way.

We start out with two fresh map variables m and m_0. The former, m, is the permission map we are going to update while constraining k_c, whereas m_0 represents the state of the permission map immediately before the method call. We let the SIL verifier assume that

Listing 12: Translation sketch for a method call involving fractional read permissions and the precondition `acc(r.val,rd)&& p ==> acc(r.val)`

```
var k_c : Permission
var m : Map[Pair[ref, Integer], Permission];
var m_0 : Map[Pair[ref, Integer], Permission];
assume 0 < k_c && 1000*k_c < k_m;
assume m == m_0;
// acc(r.val,rd)
assume m_0[(r,1)] == perm(r.val);
assert 0 < m[(r,1)];
assume k_c < m[(r,1)];
m[(r,1)] := m[(r,1)] - k_c;
// p ==> acc(r.val,rd)
if(p){
  assume m_0[(r,1)] == perm(r.val);
  assert 0 < m[(r,1)];
  assume k_c < m[(r,1)];
  m[(r,1)] := m[(r,1)] - k_c;
}
// finally, the actual call
call () := m(r,p,k_c);
```

the two maps are identical initially and later add more information about m's initial state by providing assumptions about m_0.

After k_c is sufficiently constrained, we just emit a call to our target method. The SIL verifier will have to exhale the precondition (giving away the permissions it mentions), havoc heap locations that the caller has lost all permissions to, then inhale the postcondition (receiving permissions it mentions) and finally assign results to local variables as necessary.

3.2 Asynchronous method calls (Fork-Join)

- Explain translation of fork-join.
- Explain how read-fraction tracking works identically to the synchronous case
- How context of fork is captured for use in join
- Problems with old(.) expressions.

At this time, SIL only provides synchronous call statements. We therefore have to fall back to just exhaling the precondition on fork and inhaling the postcondition on join. The challenging aspect of verifying asynchronous method calls is establishing the link between a join and the corresponding fork. Old expressions, in particular, are difficult to capture in SIL without a dedicated call statement.

k_m : read fraction selected for the surrounding scope
e : terms
P, Q : expressions
p, q : permission amount
f : field
n : integer
g : function
G : precondition of g
$x_1, \ldots, x_n$: parameters of g
$m[e]$: look up map entry with key e in map m
$\mathrm{P}[x/e]$: P, but with e substituted for x

Figure 1: Meaning of names used below.

$$\mathrm{E}[\![\texttt{acc}\,(e.f, p)]\!]_{\mathrm{Ch}} = [\![\textbf{perm}\,(\mathrm{E}[\![e]\!]_{\mathrm{Ch}}, f) < \mathrm{E}[\![p]\!]_{\mathrm{Ch}}]\!]_{\mathrm{SIL}}$$

$\mathrm{E}[\![e]\!]_{\mathrm{Ch}}$ translates expression e to SIL terms and expressions

$$\mathrm{R}[\![\mathrm{P}]\!]_{\mathrm{Ch}} = [\![$$
$$\quad \textbf{var}\ k_c, m, m_0;$$
$$\quad \textbf{inhale}\ 0 < k_c \wedge k_c * 1000 < k_m;$$
$$\quad \textbf{inhale}\ m = m_0;$$
$$\quad \mathrm{T}[\![\mathrm{P}]\!]_{\mathrm{Ch}}]\!]_{\mathrm{SIL}}$$

$$\mathrm{T}[\![\mathrm{P} \wedge \mathrm{Q}]\!]_{\mathrm{Ch}} = [\![\mathrm{T}[\![\mathrm{P}]\!]_{\mathrm{Ch}};\ \mathrm{T}[\![\mathrm{Q}]\!]_{\mathrm{Ch}}]\!]_{\mathrm{SIL}}$$

$$\mathrm{T}[\![e \Rightarrow \mathrm{Q}]\!]_{\mathrm{Ch}} = [\![\textbf{if}\,(\mathrm{E}[\![e]\!]_{\mathrm{Ch}})\,\{\mathrm{T}[\![\mathrm{Q}]\!]_{\mathrm{Ch}}\}]\!]_{\mathrm{SIL}}$$

$$\mathrm{T}[\![\texttt{acc}\,(x.f, p)]\!]_{\mathrm{Ch}} = \text{if } \mathrm{H}(\mathrm{E}[\![p]\!]_{\mathrm{Ch}}, \text{false}) \quad [\![$$
$$\quad \textbf{exhale}\ \mathrm{D}[\![x]\!]_{\mathrm{Ch}};$$
$$\quad \textbf{inhale}\ m_0\,[(\mathrm{E}[\![x]\!]_{\mathrm{Ch}}, f)] = \textbf{perm}\,(\mathrm{E}[\![e]\!]_{\mathrm{Ch}}, f);$$
$$\quad \textbf{exhale}\ 0 < m\,[(\mathrm{E}[\![x]\!]_{\mathrm{Ch}}, f)];$$
$$\quad \textbf{inhale}\ \mathrm{E}[\![p]\!]_{\mathrm{Ch}} < m\,[(\mathrm{E}[\![x]\!]_{\mathrm{Ch}}, f)];$$
$$\quad m\,[(\mathrm{E}[\![x]\!]_{\mathrm{Ch}}, f)] := m\,[(\mathrm{E}[\![x]\!]_{\mathrm{Ch}}, f)] - \mathrm{E}[\![p]\!]_{\mathrm{Ch}}]\!]_{\mathrm{SIL}}$$

$$\mathrm{T}[\![\texttt{acc}\,(x.f, p)]\!]_{\mathrm{Ch}} = \text{otherwise} \quad [\![\texttt{skip}]\!]_{\mathrm{SIL}}$$

Figure 2: Translation schemes. R generates code that constrains a fresh k_c according to the method precondition/loop invariant P. T recursively translates P to constrain k_c.

$$
\begin{aligned}
\mathrm{H}(p * q, h) &= \mathrm{H}(p, h) \wedge \mathrm{H}(q, h) \\
\mathrm{H}(p + q, h) &= \mathrm{H}(p, h) \wedge \mathrm{H}(q, h) \\
\mathrm{H}(p - q, h) &= \mathrm{H}(p, h) \wedge \mathrm{H}(q, \neg h) \\
\mathrm{H}(p * n, h) &= \mathrm{H}(p, h) \\
\mathrm{H}(k_m, h) &= \neg h \\
\mathrm{H}(\_, h) &= \text{false}
\end{aligned}
$$

Figure 3: Helper function that determines whether a permission amount expression can be used to constrain k_c.

$$
\begin{aligned}
\mathrm{D}[\![\mathrm{P} \Rightarrow \mathrm{Q}]\!]_{\mathrm{Ch}} &= [\![\mathrm{D}[\![\mathrm{P}]\!]_{\mathrm{Ch}} \wedge (\mathrm{E}[\![\mathrm{P}]\!]_{\mathrm{Ch}} \Rightarrow \mathrm{D}[\![\mathrm{Q}]\!]_{\mathrm{Ch}})]\!]_{\mathrm{SIL}} \\
\mathrm{D}[\![\texttt{if}\ (e)\ \mathrm{P}\ \texttt{else}\ \mathrm{Q}]\!]_{\mathrm{Ch}} &= [\![\mathrm{D}[\![e]\!]_{\mathrm{Ch}} \wedge (\mathrm{E}[\![e]\!]_{\mathrm{Ch}} \Rightarrow \mathrm{D}[\![\mathrm{P}]\!]_{\mathrm{Ch}}) \\
&\qquad \wedge (\neg \mathrm{E}[\![e]\!]_{\mathrm{Ch}} \Rightarrow \mathrm{D}[\![\mathrm{Q}]\!]_{\mathrm{Ch}})]\!]_{\mathrm{SIL}} \\
\mathrm{D}[\![e.f]\!]_{\mathrm{Ch}} &= [\![\mathrm{D}[\![e]\!]_{\mathrm{Ch}} \wedge \neg (\mathrm{E}[\![e]\!]_{\mathrm{Ch}} = \texttt{null}) \\
&\qquad \wedge 0 < \texttt{perm}(\mathrm{E}[\![e]\!]_{\mathrm{Ch}}, f)]\!]_{\mathrm{SIL}} \\
\mathrm{D}[\![e.g\ (a_1, \ldots, a_n)]\!]_{\mathrm{Ch}} &= [\![\mathrm{D}[\![e]\!]_{\mathrm{Ch}} \wedge \neg (\mathrm{E}[\![e]\!]_{\mathrm{Ch}} = \texttt{null}) \\
&\qquad \wedge \mathrm{E}[\![\mathrm{G}\,[\texttt{this}/e, x_1/a_1, \ldots, x_n/a_n]]\!]_{\mathrm{Ch}}]\!]_{\mathrm{SIL}}
\end{aligned}
$$

Figure 4: Translation scheme for ensuring the definedness of an expression.

$$
\begin{aligned}
&\mathrm{F}[\![\texttt{old}\ (e)]\!]_{\mathrm{Ch}} = [\![\\
&\qquad \texttt{inhale}\ \texttt{acc}\,(t.f_e, \texttt{write}) \\
&\qquad \texttt{if}\,(\mathrm{D}[\![e]\!]_{\mathrm{Ch}})\,\{ \\
&\qquad\quad t.f_e := \mathrm{E}[\![e]\!]_{\mathrm{Ch}} \\
&\qquad \} \\
&\mathrm{F}[\![\mathrm{P}]\!]_{\mathrm{Ch}} = \text{descend into P to find all } \texttt{old} \text{ expressions and terms}
\end{aligned}
$$

Figure 5: Assign $\texttt{old}(e)$ value to its own field f_e, if e is defined.

3.2.1 Translation of `fork`

The translation of the Chalice `fork` statement seems, at least at first, relatively straightforward: exhale the method's precondition and create a token object with a boolean field called "`joinable`" set to `true`. But how would we then translate the corresponding `join` statement(s)? The method's postcondition is formulated in terms of the method's return values and parameters. In general we no longer have access to the latter. The `join` might happen in a different method, but even if it occurs in the same method as the `fork`, the heap and the values of local variables could have changed in the meantime. Ideally, we could somehow capture the entire program state and store it in or associate it with the token at the `fork` statement. At the `join` statement, we would then evaluate (inhale) the method's postcondition in terms of that program state.

Sadly, SIL currently has no such mechanism. It does have `old` expressions but they are hardwired to refer to the pre-state of the surrounding method (the state immediately prior to a call to that method). Fortunately, we don't actually need to capture the entire program heap. The set of values that might be missing at the `join` site are the arguments and the values of old expressions. Since this includes at most all arguments and old expression, we can generate a ghost field on the token to "transport" each of values from the `fork` site to the `join` site.

Chalice2SIL generates one ghost field for each method argument and one ghost field for each `old` expression in the method's postcondition. Just before the `exhale` statement of a `join`, it assigns the effective arguments to the argument ghost fields of the token. It then evaluates the old expressions of the method's postcondition and assigns the results to the corresponding ghost fields.

There is just one more complication to take care of: `old` expressions can appear on the right-hand-side of implications, where they might only be defined part of the time (missing permissions and null references). Unfortunately, just expanding implications into if-statements, like we did when constraining k_c, is not an option because the left-hand-side of the implication could be a return value, which is of course only available at the `join` site. Instead, we walk over each `old` expression and generate a set of conditions that need to be satisfied for the expression to be defined at the `fork` site.

formal-ish set of rules for the "defined-ness-condition" (good name for that?)

$$\mathrm{D}[\![x.f]\!]_{\mathrm{SIL}} = \mathrm{D}[\![x]\!]_{\mathrm{SIL}} \wedge [\![x \neq \texttt{null}]\!]_{\mathrm{SIL}} \wedge [\![\texttt{perm}\,(x.f)]\!]_{\mathrm{SIL}}$$

Need to mention that k_c is computed in exactly the same way as for synchronous calls?

3.2.2 Translation of `join`

With most of the hard work done when the thread was forked, the translation of a `join` statement is relatively straightforward. First, we must assert that the token is still `joinable` (we also need write-access to that field in order to set it to false). Then we inhale the method's postcondition using the ghost fields on the token as substitutions for the arguments and old expressions. Finally, we have to assign the results of the asynchronous computation to the variables indicated by the Chalice programmer.

A detail worth mentioning is the representation of results for the `inhale` statement. Chalice2SIL also creates ghost fields on the token for results. Since a token is only ever joined

Listing 13: Example of Chalice program featuring fork and join of method with a possibly undefined **old** expression.

```
class Cell { var f : int; }
class SuperCell { var cell : Cell; }

class Main {
    method parallel(d : SuperCell) returns (r: bool)
        requires d != null ==> acc(d.cell, rd) && d.cell != null
                             && acc(d.cell.f, rd) && d.cell.f == 5
        ensures r == (d != null)
        ensures r ==> old(d.cell.f == 5)
        ensures r ==> (acc(d.cell, rd) && acc(d.cell.f, rd))
    {
          r := d != null;
    }

    method main(d : SuperCell, c : Cell)
        requires acc(d.cell) && acc(c.f)
        ensures acc(d.cell) && acc(c.f)
    {
        var r : bool;
        d.cell := c;
        c.f := 5;
        fork tk := parallel(d)
        assert c.f == 5; // still have read-access
        join r := tk;
        assert r;
    }
}
```

Listing 14: Translation of the `fork` statement on line 22 in listing 13.

```
var tk : ref;
tk := new ref;
inhale acc(tk.joinable,write);
tk.joinable := true;
// constrain k_c, the read fraction for this call
...
// store arguments in token
inhale acc(tk.this,write);
tk.this := this;
inhale acc(tk.d,write);
tk.d := d;
inhale acc(tk.k_m,write);
tk.k_m := k_c;
//store old values in token
inhale acc(tk.old1,write);
if(d != null && 0 < perm(d.cell) && d.cell != null && 0 < perm(d.cell.f)){
  tk.old1 := (d.cell.f == 5);
}
// "perform" the asynchronous call by exhaling the callee's precondition
exhale this != null && 0 < k_c && k_c < write &&
       d != null ==> acc(d.cell, k_c) && d.cell != null
                  && acc(d.cell.f, k_c) && d.cell.f == 5
```

Listing 15: Translation of the `join` statement on line 24 in listing 13.

```
exhale tk.joinable // SIL verifier also needs to assert that tk != null
inhale acc(tk.r,write) && tk.r == (tk.d != null)
   &&  tk.r ==> tk.old1
   &&  tk.r ==> acc(tk.d.cell, tk.k_m) && acc(tk.d.cell.f, tk.k_m);
r := tk.r;
tk.joinable := false;
```

once, we can safely inhale the permissions to access those result fields. Conceptually, by joining with the current thread, the forked thread transfers access to its results along with all other permissions from its postcondition.

Alternatively, we could have used fresh local variables to represent result values. The only advantage that ghost fields provide, is that we *don't* need to introduce new variables.

The accessibility of all the other ghost fields on the token requires a bit more work. Naturally, tokens can also be passed to other threads and joined there. The requirement that the joining thread has exclusive access to the `joinable` field ensures that only one thread can join on a given token. Now, while the ghost fields on the token might be invisible to the Chalice programmer, SIL does not distinguish between ghost fields and ordinary fields in any way. We need to make sure that every method that tries to access any of the ghost fields actually has permissions to do so.

Fortunately, ghost fields on a token are only accessed when we also have permission to access the `joinable` field on that token and it is the Chalice programmer's burden to ensure that a thread has this permission when attempting to join on a token. If we could somehow link the amount of permission a thread has to each of the ghost fields to the amount of permission it holds to `joinable`, we would always end up with a sufficient amount of permission for the ghost fields.

While SIL provides no built-in support for linking fields together accessibility-wise, we can achieve a similar effect by translating every accessibility predicate for `joinable` as an accessibility predicate for that *and* all ghost fields (with the same amount of permission for each). That way, we can be sure that whenever a thread holds full permissions to a `joinable` field, it also holds full permissions to all ghost fields on the token. More formally, given a token t, a permission amount p, ghost fields $a_1 \cdots a_k$ (the arguments) and $o_1 \cdots o_n$ (evaluated old expressions), we apply the following transformation:

$$[\![\texttt{acc}\,(t.\texttt{joinable}, \mathrm{p})]\!]_{\mathrm{SIL}}$$

becomes

$$[\![\texttt{acc}\,(t.\texttt{joinable}, p) \wedge \texttt{acc}\,(t.\texttt{this}, p) \wedge$$
$$\wedge\ \texttt{acc}\,(t.a_1, p) \wedge \texttt{acc}\,(t.a_2, p) \wedge \cdots \wedge \texttt{acc}\,(t.a_k, p) \wedge$$
$$\wedge \texttt{acc}\,(t.o_1, p) \wedge \texttt{acc}\,(t.o_2, p) \cdots \texttt{acc}\,(t.o_n)]\!]_{\mathrm{SIL}}$$

3.2.3 Limitations of the current fork-join implementation

Joining a thread seems deceptively simple when done in the same method the thread was originally forked from. This is because the verifier has seen the assignments to the token ghost fields first hand. When a thread is joined in a separate method, however, that context is not available because both Silicon and the Boogie-based implementation verify each method in complete isolation.

For just joining a thread in a separate method, the programmer needs to pass both the token and write access to the token's `joinable` field to the method that performs the joining and ensure that the thread has not been joined already. Unfortunately, the postcondition of an asynchronous method call joined this way is next to useless, because the verifier has no information about the context of the method call. Specifically, the verifier doesn't know anything about the receiver or any of the arguments originally passed to the thread. As a consequence, any clause of the postcondition that mentions the `this` pointer or an argument is useless to the verifier.

Listing 16 demonstrates a simple program that fails to verify because the context of the forked thread is lost when the token is transferred to the callee (`client`). The verifier will complain that there might not be enough permission to satisfy `acc(obj.f)`, because it doesn't know that the `this` pointer used to call `work` refers to the same object as `obj`. We would like to tell the verifier more about how our token was created.

```
        requires tk.thisPtr == obj //not valid Chalice expression
```

While the previous example is not valid Chalice code, there is a mechanism that can be used to create similar specifications. Listing 17 shows how the `eval` expression can be

Listing 16: Limitations with joining in separate methods

```
class Main{
    var f : int;
    method work()
        requires acc(this.f)
        ensures acc(this.f)
    {
    }

    method main()
        requires acc(this.f)
        ensures acc(this.f)
    {
        fork tk := work();
        call client(tk, this);
    }

    method client(tk : token<Main.work>, obj : Main)
        requires acc(tk.joinable) && tk.joinable
        ensures acc(obj.f) // might not hold
    {
        join tk;
    }
}
```

Listing 17: **eval** expression in Chalice

```
method client(tk : token<Main.work>, obj : Main)
        requires acc(tk.joinable) && tk.joinable
        requires eval(tk.fork this.work(), this == obj)
        ensures acc(obj.f)
    { join tk; }
```

used to provide the verifier with the information necessary to prove that the method satisfies its postcondition.

An $[\![\texttt{eval}\,(r.a, e)]\!]_{\text{Ch}}$ expression consists of three parts: the "context" c (the token in our case), the description of the "*eval state*" a and an expression e to be evaluated in that state. In our case, we specify a "*call state*" of the form $[\![\texttt{fork}\,r.m\,(a_1, a_2, \cdots, a_k)]\!]_{\text{Ch}}$. Here r denotes the receiver of the asynchronous method call, m is the name of the method called and a_i stand for the arguments originally passed to the method.

- why `eval` expr was not implemented

3.3 Predicates and Functions

- 1:1 correspondence between Chalice and SIL
- Explain global predicate- and function-Fractions

3.4 Monitors with Deadlock Avoidance

- Explain global monitor fraction
- Too high-level for Mu: why establishing the correspondence between x.mu and waitlevel is difficult
- Explain the solution: muMap, heldMap and the $CurrentThread object.

4. Evaluation

4.1 Chalice2SIL+Silicon versus Sy-whatever

Compare Chalice2SIL+Silicon performance with Syxc (?) both in terms of execution speed and capabilities

4.2 SIL as a translation target/verification intermediate language

- Term/Expression distinction (technical)
- P-expressions/terms versus non-P-expressions/terms (technical)
- control flow (need for a while loop construct)
- Some way of capturing state (more general old expression, "transaction"-like scopes)

5. Conclusion

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.