Translating Chalice into SIL

Problem Description

Christian Klauser
klauserc@student.ethz.ch

August 4, 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

- Goals/general ideas of Chalice: Permission-based, modular
- Inhale, Exhale, Havoc, terms used in Boogie implementation
- Methods
- Fork+Join
- Predicates + Functions
- Monitors

2.2 Semper Intermediate Language (SIL)

3. Translation of Chalice

High-level overview + including focus areas

3.1 Fractional Read Permissions

To SIL, permissions are just another data type. The SIL prelude only defines a set of constructors (like no permission, full permission) and some operators and predicates (like

Listing 1: 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;
}
```

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. Like 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)` do usually not represent the same amount of permission.

This makes fractional permissions 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.

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.

Listing 2: 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
}
```

Listing 3: 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)
{ … }
```

For verifying the callee in listing 1, 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`. Of course, k_m remains constant throughout the entire body of a method.

Notice how the Boogie-based encoding of Chalice in listing 2 does not make use of the pre- and postcondition mechanism provided by Boogie. This is primarily because Boogie 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. 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.

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.

```
method m(r : Register, p : bool)
  requires acc(r.val, rd) && (p ==> acc(r.val, rd))
  ...
```

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.

Listing 4: 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);
```

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

Formal-ish translation rules for method call here?

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

$$\mathrm{T}\,[\![\mathrm{P} \Rightarrow \mathrm{Q}]\!]_{\mathrm{Ch}} = [\![\mathtt{if}\,(\mathrm{T}\,[\![\mathrm{P}]\!]_{\mathrm{Ch}})\,\{\mathrm{T}\,[\![\mathrm{Q}]\!]_{\mathrm{Ch}}\}]\!]_{\mathrm{SIL}}$$

etc.

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.

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 for use in postconditions, but they only carry a special meaning in conjunction with the synchronous SIL `call` statement. Fortunately, we don't actually need to capture the entire program heap. The set of values missing at the `join` site are the arguments and the values of old expressions. Since the size of this set is constant and known statically, we can use ghost fields on the token to "transport" these 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

Listing 5: Example of 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;
    }
}
```

of the implication could be a return value. 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.

Listing 6: Translation of the `fork` statement on line 22 in listing 5.

```
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 7: Translation of the `join` statement on line 24 in listing 5.

```
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;
```

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 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 8 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
```

Listing 8: 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 9: `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; }
```

While the previous example is not valid Chalice code, there is a mechanism that can be used to create similar specifications. Listing 9 shows how the `eval` expression can be 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