Translating Chalice into SIL

Problem Description

Christian Klauser
klauserc@student.ethz.ch

July 25, 2012

1. Introduction

2. Semper Intermediate Language (SIL)

3. Translation of Chalice

High-level overview + including focus areas

3.1 Fractional Read Permissions

High-level description of how read-fractions are translated in Boogie
SIL call node versus same approach as Boogie-encoding
Explain how read-permissions are translated

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 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.

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

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;
}
```

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)
{ … }
```

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.

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.

```
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;
}
call () := m(r,p,k_c);
```

3.2 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.

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

5. Conclusion