Translating Chalice into SIL

Bachelor Thesis Report

Chair of Programming Methodology
Department of Computer Science
ETH Zürich
www.pm.inf.ethz.ch

By: Christian Klauser
klauserc@student.ethz.ch

Supervised by: Dr. Alexander J. Summers
Prof. Dr. Peter Müller

Date: November 29, 2012

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Contents

1 Introduction

Writing correct computer programs is difficult. Writing correct concurrent or parallel computer programs is even more difficult. One approach to making sure programs do not contain errors is (automatic) *static verification*: the idea of having a computer prove that a given program fulfils its specification and does not crash. An example of such a system is *Chalice* [LMS09] (section 2.1), a research programming language and matching automatic static verification tool. However, targeting a specialized research language dedicated to the verification of concurrent programs, means that one cannot directly apply the tool to code that is used out in the world.

This is where *Semper*, a project at ETH Zürich, comes into play. Its goal is to develop an automatic program verifier for concurrent *Scala* [Sca12] programs. Central to the Semper project is an intermediate program representation for verification called *SIL* (section 2.2). Programmers are not intended to use SIL directly, but instead write their programs in an existing programming language and then use a translator to get an intermediate representation that the Semper tools understand.

The goal of this Bachelor's thesis is to build *Chalice2SIL*, the first such translator, translating from Chalice to SIL (section 3), in order to gain experience with working with SIL (section 4) and the tools involved in Semper. As the verification methodology used in Semper is based on the methodology underlying Chalice, Chalice is a good fit for the first "source language" to be targeted by Semper.

2 Background

This section briefly presents both *Chalice* (the "source language") and *SIL* (the "target language"), focusing on the aspects that are important for discussing how Chalice2SIL performs its translation.

2.1 Chalice

Chalice [LMS09, LM09] 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#) [BLS], 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 [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.

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;
    }
  }
}
```

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 every iteration, including the last one, when 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. A thread having *write* permission to a heap location means that that thread holds *all* permissions to that location. However, permissions can also be divided up between multiple threads and as long as a thread holds onto a strictly positive amount of permission to a heap location, it can not only read that location, but is also guaranteed that no thread can write to that location, as write-access requires 100% of all permissions to a heap location. For a thread to have no permissions at all to a heap location means that this location is completely inaccessible. Worse yet, it could be changing at any moment, since another thread might hold full permissions to it. Chalice therefore forbids access to heap locations where the current thread holds no permissions to.

The amount of permission a thread holds over a certain heap location can change over time, and often does. The main thread of an implementation of a parallel algorithm, for instance, could start with exclusive write-access to the input data structure and then split up its permissions among a number of worker threads. These worker threads could then

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;
  }
}
```

all *read* from the input data structure while performing their work, relying on the fact that concurrent accesses to that data structure are safe since no thread can write to it. After the worker threads have finished, the main thread can collect the permission fractions given out to the workers and combine them back to a full permission, allowing it to write to the input data structure once again, perhaps to update it with the results retrieved from the worker threads.

Almost as a side-effect of this model, the amount of permission a thread has can be used to *"frame"* the set of existing heap locations that thread can access (a thread can always allocate new heap space). All these permissions only exist for verification and would be erased by compilers for Chalice.

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 from 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 represents a permission, which 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 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*.

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

```
class Program {
  //...
  method main()
  {
    var c : Cell :=  new Cell;
    c.f := 5;
    var d : Cell;
    call d := clone(c);
    assert d.f == 5; // will fail, c.f might have changed
  }
}
```

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. grants read and write access, whereas any other strictly positive percentage `acc(x.f,n)` (for $n \in \mathbb{N}, 0 < n < 100$) only grants read access to the heap location `x.f`. 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 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 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 [HLMS11] added an entirely new kind of permission to Chalice: the fractional read permission, based on [Boy03]. The idea is to allow for "rational" fractions of permission because, unlike epsilon or percentage permissions, these can always be divided further. Composability can still be an issue, even with concrete 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 abstract 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 some permission to that location. That way, the caller retains read-access and 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.

Chalice restricts read fractions in method specifications even further: for each method invocation, all fractional read permissions in the 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 the permissions that the caller retained to determine which locations *cannot* be modified by the call.

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.

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
  }
}
```

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 }
}
```

```
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 returns 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. 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

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;
    }
}
```

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 get access to the contents of a predicate during the evaluation of an expression.

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 invariant (similarly to an **`unfold`** of a predicate). The object is now *locked* and can be made available to other threads via the **`release`** statement (similarly to a **`fold`** of a predicate, 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.

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;
    }
}
```

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 according to a consistent ordering. 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 denoted 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 declares 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 `a` and 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; this would result in an error.

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 readable in order to acquire the lock.

2.1.8 Details on the Boogie-based Chalice verifier

In order to verify Chalice programs, the Boogie-based verifier models permission transfer by two operations: **inhale** and **exhale**. They are essentially the same as **assume** and

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
    }
}
```

assert but in addition to providing and checking facts, they also model the transfer of permissions. The argument of an **exhale** operation is an expression that can contain both traditional (boolean) assertions as well as accessibility predicates. Conceptually, **exhale** *e* represents the transfer of *e* to another thread. Because verification of Chalice methods is modular, we don't specify or even care about which thread will "receive" *e*. For each **exhale** *o*peration, the verifier will check (assert) the boolean predicates and remove permissions mentioned in *e* from the current thread's set of permissions (usually referred to as the thread's "*permission mask*"). The **inhale** *e* operation works the opposite way. Access permissions mentioned in *e* are added to the thread's permission mask and boolean predicates get assumed. More advanced features such as method calls and monitors are translated into combinations of **inhale** , **exhale** , **assume** and **assert** operations.

2.2 Semper Intermediate Language (SIL)

The Semper Intermediate Language is a verification language aimed at the verification of concurrent programs using a methodology based on Chalice. As its name suggests, SIL is the intermediate language to be used by the various tools that are part of the semper project.

Much of SIL's design is oriented around Chalice's core elements: methods, permissions and accessibility predicates. This also means that SIL programs are encoded on a much higher level of abstraction than the same programs in less focused verification languages, such as Boogie. As an example: the Boogie-based verifier for Chalice needs to represent permissions as a pair of integers (the number of epsilons and the percentage) whereas in SIL there is a dedicated and built-in data type and associated value constructor functions for permissions.

In this section, we will give an overview of the syntactic structure of SIL programs, diving

into more detail where the design of SIL deviates significantly from Chalice. At this time SIL is mostly intended as an "exchange format" and thus has no fixed semantics associated with it. Also, SIL doesn't currently have a serialised/text form and SIL programs only exist as syntax trees in memory. As a result we use our own ad-hoc textual representation for SIL program snippets in this report.

2.2.1 SIL Program Structure

Each SIL program has a name (⟨*program-id*⟩) and comes with a number of *domain*, *field*, *function*, *predicate* and *method definitions*. While SIL is certainly aimed at the verification of object oriented programs, it isn't actually necessary to distinguish between the types of references to objects created from different classes. As a direct result, fields, functions, predicates and methods are not "contained" in any form of class definition.

⟨*Program*⟩ ::= 'program' ⟨*program-id*⟩
{⟨*Domain*⟩}
{⟨*Field*⟩}
{⟨*Function*⟩}
{⟨*Predicate*⟩}
{⟨*Method*⟩}

Field and predicate definitions, apart from the fact that they are not tied to a nominal class, are fairly straightforward. Fields consist of a name and a data type and predicates consist of a name and an expression. As with Chalice, this predicate expression can contain both accessibility predicates and ordinary boolean predicates. Field and predicate names must each be unique within a SIL program.

⟨*Field*⟩ :: 'field' ⟨*field-id*⟩ ':' ⟨*DataType*⟩

⟨*Predicate*⟩ ::= 'predicate' ⟨*pred-id*⟩ '=' ⟨*Expr*⟩

Functions, again, are similar to their Chalice counterparts. They consist of a name, a parameter list, a result type, some preconditions and an implementation. Note how an ⟨*Expr*⟩ is expected for the preconditions and a ⟨*Term*⟩ for the function's body. This is an example of SIL distinguishing syntactically between assertions/formulae (⟨*Expr*⟩) and expressions that represent a value (⟨*Term*⟩).

⟨*Function*⟩ ::= 'function' ⟨*id*⟩ ({ ⟨*Param*⟩ , ⋯ }) : ⟨*DataType*⟩
⟨*Contract*⟩ '=' ⟨*Term*⟩

⟨*Param*⟩ ::= ⟨*id*⟩ : ⟨*DataType*⟩

⟨*Contract*⟩ ::= { 'requires' ⟨*Expr*⟩ } { 'ensures' ⟨*Expr*⟩ }

Methods in SIL have a name (unique among all methods in the program), input and output parameters and a set of pre- and postconditions. Every SIL method always has a parameter called `this` of type **`ref`** in the first position, which represents the **`this`** pointer in object oriented languages. Having the `this` pointer as an ordinary parameter makes tools that consume SIL programs a bit simpler. Each method can have multiple implementations that must all share the exact same parameters, pre- and postconditions. For source languages with virtual methods, the to-SIL-translator would create a method for each "method slot" (vtable slot) and add an implementation for each concrete implementation encountered in the program.

⟨*Method*⟩ ::= 'method' ⟨*method-id*⟩ ({ ⟨*Param*⟩ , ⋯ }) : ({ ⟨*Result*⟩ , ⋯ })
⟨*Contract*⟩ { ⟨*Impl*⟩ }

⟨*Result*⟩ ::= ⟨*Param*⟩

⟨*Impl*⟩ ::= 'implementation' ⟨*method-id*⟩ ⟨*Cfg*⟩

Method bodies in SIL are represented as a control-flow graph. This is mostly because SIL is intended as a format for exchanging programs between the tools that make up Semper as opposed to an actual computer languages used by humans. Whether an eventual textual representation would retain this form, is not clear at this point.

Unusual about SIL's control-flow graph is that loops are not flattened into basic blocks but retained as a sort of composite block. A loop block consists of the loop condition, an invariant and a nested control-flow graph for the loop's body.

⟨*Cfg*⟩ ::= '{' { ⟨*VarDecl*⟩ } { ⟨*Block*⟩ } '}'

⟨*VarDecl*⟩ ::= 'var' ⟨*var-id*⟩ : ⟨*DataType*⟩

⟨*Block*⟩ ::= ⟨*BasicBlock*⟩
| ⟨*LoopBlock*⟩

⟨*LoopBlock*⟩ ::= 'while' ⟨*PExpr*⟩ ['invariant' Expr] 'do' ⟨*Cfg*⟩

⟨*BasicBlock*⟩ ::= ⟨*label*⟩: '{' { ⟨*Stmt*⟩ } ⟨*ControlFlow*⟩ '}'

⟨*ControlFlow*⟩ ::= 'goto' ⟨*label*⟩
| 'halt'
| 'if' ⟨*PExpr*⟩ 'then goto' ⟨*label*⟩ 'else goto' ⟨*label*⟩

At the end of every block there is a single control-flow statement that indicates how control is transferred to other blocks.

2.2.2 SIL Statements

⟨*Stmt*⟩ ::= ⟨*var-id*⟩ ':=' ⟨*PTerm*⟩
| ⟨*var-id*⟩.⟨*field-id*⟩ ':=' ⟨*PTerm*⟩
| ⟨*var-id*⟩ ':= new' ⟨*DataType*⟩
| ⋮

⟨*Stmt*⟩ ::= ⋮
| ({ ⟨*var-id*⟩ , ⋯ }) ':=' ⟨*PTerm*⟩.⟨*method-id*⟩({ ⟨*PTerm*⟩ , ⋯ })
| ⋮

⟨*Stmt*⟩ ::= ⋮
| 'inhale' ⟨*Expr*⟩
| 'exhale' ⟨*Expr*⟩
| ⋮

⟨*Stmt*⟩ ::= ⋮
| 'fold' ⟨*Term*⟩.⟨*pred-id*⟩ 'by' ⟨*Term*⟩
| 'unfold' ⟨*Term*⟩.⟨*pred-id*⟩

2.2.3 SIL Expressions and Terms

⟨*Expr*⟩ ::= 'acc' (⟨*Location*⟩, ⟨*Term*⟩)
| 'old' (⟨*Expr*⟩)
| 'unfolding' ⟨*Term*⟩.⟨*pred-id*⟩ 'by' ⟨*Term*⟩ 'in' ⟨*Expr*⟩
| ⟨*Term*⟩ == ⟨*Term*⟩
| ⟨*unary-op*⟩ ⟨*Expr*⟩
| ⟨*binary-op*⟩ ⟨*Expr*⟩
| ⟨*dom-pred-id*⟩({ ⟨*Term*⟩ , ⋯ })
| ∀ ⟨*logical-var-id*⟩ : ⟨*DataType*⟩ :: (⟨*Expr*⟩)
| ∃ ⟨*logical-var-id*⟩ : ⟨*DataType*⟩ :: (⟨*Expr*⟩)

⟨*Location*⟩ ::= ⟨*Term*⟩.⟨*field-id*⟩
| ⟨*Term*⟩.⟨*pred-id*⟩

⟨*Term*⟩ ::= 'if' ⟨*Term*⟩ 'then' ⟨*Term*⟩ 'else' ⟨*Term*⟩
| ⟨*var-id*⟩
| ⟨*logical-var-id*⟩
| 'old'(⟨*Term*⟩)
| ⟨*func-id*⟩({ ⟨*Term*⟩ , ⋯ })
| ⟨*dom-func-id*⟩({ ⟨*Term*⟩ , ⋯ })
| 'unfolding' ⟨*Term*⟩.⟨*pred-id*⟩ 'by' ⟨*Term*⟩ 'in' ⟨*Term*⟩
| (⟨*Term*⟩) : ⟨*DataType*⟩
| ⟨*Term*⟩.⟨*field-id*⟩
| 'perm'(⟨*Location*⟩)
| 'write'
| '0'
| 'E'
| ⟨*integer-literal*⟩

We simplified the presentation of the term and expression grammar for this section and attached the full rules in appendix A.

2.2.4 SIL Domains and Types

A data type in SIL is either '`ref`', the type of all object references, a domain type or a type variable. Object references in SIL are treated as potentially having all fields in the SIL program. In practice, only the fields that a method/function has access to, are relevant. For statically typed programming languages, it's the responsibility of the to-SIL-translator to make sure that input programs are type error free.

⟨*DataType*⟩ ::= ⟨*var-type*⟩
| ⟨*dom-type*⟩
| 'ref'

In addition to the built-in value domains for integers, booleans and permissions, SIL allows its users to define their own value domains, with (uninterpreted) constructor functions, predicates over values of that domain and their axioms. Domain definitions can come with type parameters, making them templates for concrete domains (similar to C# generics).

⟨*Domain*⟩ ::= 'domain' ⟨*dom-id*⟩ [⟨*DomainParameters*⟩] '{' ⟨*DomainDef* ⟩ '}'

⟨*DomainDef*⟩ ::= { ⟨*DomainFunction* ⟩} { ⟨*DomainPredicate*⟩ } { ⟨*DomainAxiom*⟩ }

⟨*DomainFunction*⟩ ::= 'function' ⟨*dom-func-id*⟩ ({ ⟨*DataType*⟩ , ⋯ }) : ⟨*DataType*⟩

⟨*DomainPredicate*⟩ ::= 'predicate' ⟨*dom-pred-id*⟩ ({ ⟨*DataType*⟩ , ⋯ })

⟨*DomainAxiom*⟩ ::= 'axiom' ⟨*id*⟩ '=' ⟨*DExpr*⟩

⟨*DomainParameters*⟩ ::= '[' { ⟨*DataType*⟩ , ⋯ } ']'

2.3 Silicon

Silicon is an automated program verifier for SIL programs based on symbolic execution. It was derived from Syxc [Sch11], an alternative verifier for Chalice, and adapted to verify SIL instead. As Silicon is currently the only verifier for SIL, we use it to test Chalice2SIL.

3 Translation of Chalice

This section explains how Chalice2SIL translates the most interesting aspects of Chalice into SIL. As Chalice and SIL code often look very similar, we use $[\![e_1]\!]_{\text{Ch}}$ to emphasize the fact that e_1 is a Chalice expression or statement. Similarly $[\![e_2]\!]_{\text{SIL}}$ stands for the SIL expression e_2.

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 accessibility 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 b ==> acc(x.f, rd)
```

as

```
if(b)
{
  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 subsequent call statement would fail. We cannot simply create a copy of the permission mask as a whole and have exhale operate on that instead because SIL considers the permission mask an implementation detail and thus doesn't expose it. SIL at least allows us to look up individual entries of the permission mask via the `perm(x.f)` term. `perm(x.f)` represents the amount of permission the current thread holds for the location $x.f$. We use that feature to manually create and maintain a permission mask 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.

As listing 12 demonstrates, 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 (line 5) and later add more information about m's initial state by providing assumptions about m_0.

The first accessibility predicate we translate is `acc(r.val,rd)`. On line 7 we copy the amount of permission we currently hold to `r.val` into our own permission map. The key, `(r,1)`, is a pair consisting of the object reference `r` and a unique integer that Chalice2SIL assigned to the field `Register::val`. No other field in the program shares that integer identifier.

We then continue with ensuring that we still have access to that field (line 8) by asserting that the amount of permission for `r.val` is strictly positive. Only then can we use the amount of permission we hold to `r.val` as an additional upper bound to k_c (line 9), otherwise that assumption could have contradicted the assumption about k_c made on line 4. Finally, on line 10 we "simulate" exhaling the permission by subtracting it from our own permission map entry. Notice how we used an assumption about the "original" map m_0 on line 7, but then continued to use m, the actual permission map, to perform the simulation.

For the second part of the callee's precondition, `p ==> acc(r.val,rd)`, we first have to deal with the implication. In accordance with our translation scheme for the read fraction constraints (figure 3), we wrap the translation of the right-hand side of the implication in an `if`-block (line 12). As the accessibility predicate `acc(r.val,rd)` is identical to what we had before, our translation also generates the same sequence of statements.

Now the fact that we do not simply copy the amount of permission we currently hold into our working permission map m becomes essential. Using an assumption on line 13 means that m_0 still is the map from which the current version of m (we updated it on line 10) is derived. Line 15 adds an even lower upper bound to the constraints on k_c because we had already subtracted k_c from the permission amount for `r.val` on line 10.

After k_c is sufficiently constrained, we just emit a call to our target method, passing k_c as a ghost parameter. 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)

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: constrain a fresh k_c to be used as the fractional read permission amount, exactly

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);
```

k_m : read fraction selected for the surrounding scope
e : program expressions
P, Q : assertions
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
$P[x/e]$: P, but with e substituted for x

Figure 1: Meaning of names used below.

$$
\begin{aligned}
H(p * q, h) &= H(p, h) \wedge H(q, h) \\
H(p + q, h) &= H(p, h) \wedge H(q, h) \\
H(p - q, h) &= H(p, h) \wedge H(q, \neg h) \\
H(p * n, h) &= H(p, h) \\
H(k_m, h) &= \neg h \qquad k_m \text{ is abstract fraction} \\
H(p, h) &= \text{false} \qquad \text{otherwise}
\end{aligned}
$$

Figure 2: Helper function that determines whether a permission amount expression can be used to constrain k_c.

$$
\begin{aligned}
E[\![\texttt{acc}\,(e.f, p)]\!]_{\text{Ch}} &= [\![\textbf{perm}\,(E[\![e]\!]_{\text{Ch}}, f) < E[\![p]\!]_{\text{Ch}}]\!]_{\text{SIL}} \\
&\quad E[\![e]\!]_{\text{Ch}} \text{ translates expression } e \text{ to SIL terms and expressions} \\
R[\![P]\!]_{\text{Ch}} &= [\![\\
&\qquad \textbf{var}\ k_c, m, m_0; \\
&\qquad \textbf{inhale}\ 0 < k_c \wedge k_c * 1000 < k_m; \\
&\qquad \textbf{inhale}\ m = m_0; \\
&\qquad T[\![P]\!]_{\text{Ch}}]\!]_{\text{SIL}} \\
T[\![P \wedge Q]\!]_{\text{Ch}} &= [\![T[\![P]\!]_{\text{Ch}}; T[\![Q]\!]_{\text{Ch}}]\!]_{\text{SIL}} \\
T[\![e \Rightarrow Q]\!]_{\text{Ch}} &= [\![\textbf{if}\,(E[\![e]\!]_{\text{Ch}})\,\{T[\![Q]\!]_{\text{Ch}}\}]\!]_{\text{SIL}} \\
T[\![\texttt{acc}\,(x.f, p)]\!]_{\text{Ch}} &= \text{if } H(E[\![p]\!]_{\text{Ch}}, \text{false}) \quad [\![\\
&\qquad \textbf{exhale}\ D[\![x]\!]_{\text{Ch}}; \\
&\qquad \textbf{inhale}\ m_0\,[(E[\![x]\!]_{\text{Ch}}, f)] = \textbf{perm}\,(E[\![e]\!]_{\text{Ch}}, f); \\
&\qquad \textbf{exhale}\ 0 < m\,[(E[\![x]\!]_{\text{Ch}}, f)]; \\
&\qquad \textbf{inhale}\ E[\![p]\!]_{\text{Ch}} < m\,[(E[\![x]\!]_{\text{Ch}}, f)]; \\
&\qquad m\,[(E[\![x]\!]_{\text{Ch}}, f)] := m\,[(E[\![x]\!]_{\text{Ch}}, f)] - E[\![p]\!]_{\text{Ch}}]\!]_{\text{SIL}} \\
T[\![\texttt{acc}\,(x.f, p)]\!]_{\text{Ch}} &= \text{otherwise} \quad [\![\texttt{skip}]\!]_{\text{SIL}}
\end{aligned}
$$

Figure 3: Translation schemes. R generates code that constrains a fresh k_c according to the method precondition/loop invariant P. T recursively translates P to constraint k_c. $D[\![x]\!]_{\text{Ch}}$ determines whether x is well-defined. See figure 5.

$$
\begin{aligned}
F[\![\texttt{old}\,(e)]\!]_{\text{Ch}} = [\![& \\
& \texttt{inhale acc}\,(t.f_e, \texttt{write}) \\
& \texttt{if}\,(D[\![e]\!]_{\text{Ch}})\,\{ \\
& \quad \texttt{var}\ b_e : \texttt{bool}; \\
& \quad \texttt{inhale eval}(b_e) == E[\![e]\!]_{\text{Ch}}; \\
& \quad t.f_e := b_e \\
& \}]\!]_{\text{SIL}} \qquad \text{if } e \text{ is an assertion} \\
F[\![\texttt{old}\,(e)]\!]_{\text{Ch}} = [\![& \\
& \texttt{inhale acc}\,(t.f_e, \texttt{write}) \\
& \texttt{if}\,(D[\![e]\!]_{\text{Ch}})\,\{ \\
& \quad t.f_e := E[\![e]\!]_{\text{Ch}} \\
& \}]\!]_{\text{SIL}} \qquad \text{otherwise} \\
F[\![P \;\&\&\; Q]\!]_{\text{Ch}} = & [\![F[\![P]\!]_{\text{Ch}}; F[\![Q]\!]_{\text{Ch}}]\!]_{\text{SIL}} \\
F[\![P]\!]_{\text{Ch}} = & \text{ analogous}
\end{aligned}
$$

Figure 4: F recursively descends into an expression looking for old expressions **old**(e). Adds a corresponding field f_e to the token t and, if the expression is well-defined at that point, evaluates e and assigns the result to f_e.

as we did for the synchronous method call, then exhale the method's precondition and finally 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. Since the set of values that might be missing at the `join` site 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. The translation scheme F presented in figure 4 shows how this is done.

There is just one more complication to take care of: `old` expressions can appear on the

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

Figure 5: Translation scheme for ensuring the definedness of an expression.

right-hand-side of implications, where they might only be defined part of the time (due to 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 refer to 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. These are similar to the defined-ness conditions in [SJP12, p12], which also appear in the Boogie-based Chalice verifier. Figure 5 shows the most important rules for generating definedness conditions $D[\![e]\!]_{\text{Ch}}$ for an expression e.

We then use these defined-ness conditions at the **`fork`**-site to guard the assignment of the **`old`** values. Listing 14, a translation of the **`fork`** statement in listing 13, demonstrates this on line 16. The defined-ness conditions in the **`if`**-condition guarantee that the computation and assignment of the **`old`** value on line 17 cannot fail. When the verifier arrives at a point at the **`join`** site where the ghost field on the token corresponding to the **`old`** expression is read, we can be in one of two cases. Either the **`old`** expression was well defined at the **`fork`**-site or it was not. If it was, the verifier will have chosen the path involving the assignment of the token ghost field and consequently has information about the contents of that field. If the old expression was not well-defined at the **`fork`**-site, then the assignment of the ghost field will have been skipped.

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 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.

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;
```

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 some-

how 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 token 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:

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

becomes

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

3.2.3 Limitations of the current fork-join implementation

Joining a thread seems deceptively simple when done in the same method that 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 a 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 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.

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; }
```

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.

Chalice2SIL supports a very limited form of the **`eval`** expression which covers exactly the use-case outlined above. As long as the **`eval`** expression binds to a **`fork`** token and has **`true`** as its second operand e, Chalice2SIL translates it as follows:

$$
\begin{aligned}
V[\![\texttt{eval}(t.\texttt{fork}\ r.m(a_1, a_2, \cdots, a_k), \texttt{true})]\!]_{\text{Ch}} = [\![\ & E[\![t]\!]_{\text{Ch}} \neq \texttt{null} \\
& \wedge E[\![t]\!]_{\text{Ch}}.\texttt{this} = E[\![r]\!]_{\text{Ch}} \\
& \wedge E[\![t]\!]_{\text{Ch}}.a_1 = E[\![a_1]\!]_{\text{Ch}} \\
& \quad \vdots \\
& \wedge E[\![t]\!]_{\text{Ch}}.a_k = E[\![a_k]\!]_{\text{Ch}}]\!]_{\text{SIL}}
\end{aligned}
$$

This small extension is just expressive enough to associate tokens with parts of the context they were forked from, allowing the joining method to actually take advantage of the postcondition of the forked method. As the general design of the **`eval`**-expression is under discussion, it did not make sense to fully support it.

3.3 Predicates and Functions

Predicates and functions have been part of SIL since its inception, in a form that very much resembles the functions and predicates from Chalice. As a result, the translation of functions and predicates from Chalice to SIL is relatively straightforward.

3.3.1 Predicates

In contrast to abstract fractional read permissions (**`rd`**) in methods, which can assume a different value for each invocation, the fraction used in predicates remains fixed. This is essential to ensure that the predicate holds the same amount of permission regardless of where it was folded. Otherwise the user would have to specify exactly how much permission a predicate contains, which rather defeats the purpose of predicates (information hiding).

Abstract read permissions mentioned in predicates are thus interpreted by a fixed permission amount. To implement this, we define an uninterpreted constant function `globalPredicateReadFraction()` and declare that this global read fraction is strictly positive and less than the full/write permission amount.

It is possible that, in the future, we'd like to have different fractions for different predicates or even different fractions for each combination of object (this-pointer) and predicate. To support this scenario, we don't insert references to `globalPredicateReadFraction()` directly into the generated SIL program. Instead, we use an intermediate function `predicateReadFraction(int,ref)`; also uninterpreted. Analogously to the "field identifiers" that we used to index into our copy of the permission mask when we constrain the read permission fraction for method calls, we generate unique "predicate identifiers" to

distinguish between different predicates on the same object. However, at the moment every SIL program that Chalice2SIL generates also contains an axiom that makes predicate read fractions effectively global:

$$\forall i, r.\ \texttt{predicateReadFraction(i,r)} = \texttt{globalPredicateReadFraction()}$$

This constraint makes it easier to transfer abstract read permissions between predicates, giving **`acc`**`(x.f,`**`rd`**`)` a fixed meaning across all predicates. Outside of predicates, the user can refer to the same amount of permission by explicitly mentioning the predicate in an argument to the abstract read permission. Given two object references x and y, a field f, a predicate p and a corresponding predicate identifier i_p, Chalice2SIL translates an accessibility predicate that involves an abstract read fraction inside a predicate body as follows:

$$[\![\texttt{acc}\,(x.f, \texttt{rd}(y.p))]\!]_{\text{Ch}} = [\![\texttt{acc}\,(E[\![x.f]\!]_{\text{Ch}}, \texttt{predicateReadFraction}(i_p, E[\![y]\!]_{\text{Ch}}))]\!]_{\text{SIL}}$$

3.3.2 Functions

Ideally, we would want to encode abstract read permissions in functions the same way we encode them when they occur in method pre/postconditions: constraining a different fraction for each call site. Unfortunately, the technique we used for read fractions in method calls would not work for function calls, because for our solution we need to create and then destructively update our own copy of the permission mask. Function calls can occur in method pre/postconditions and even inside other function bodies. All of those contexts are pure, which means that we cannot introduce and then update new "local" variables.

But functions being pure also gives us more freedom, since we don't have to make the distinction between read permissions and read-write permissions (functions aren't allowed to modify the heap anyway). Similarly, as functions always automatically return the same amount of permission that they received, it doesn't really matter exactly "how much" permission a function has, only to which heap locations it has access. As a result, Chalice2SIL translates non-write permission amounts in function preconditions as **`rd`***, which means "any read-permission".

$$[\![\texttt{acc}\,(x.f, rd*)]\!]_{\text{Ch}} = [\![\exists a, 0 < a < \texttt{write}.\ \texttt{acc}\,(x.f, a)]\!]_{\text{SIL}}$$

3.4 Monitors with Deadlock Avoidance

As with predicates, the permissions stored in a monitor need to be fixed and cannot be chosen every time we lock an object. Otherwise we'd have to track the amount of permission the monitor holds onto at any point in the program as global state and be able to communicate these permission amounts in method pre- and postcondition. Using a fixed fractional read permission is how [HLMS11] implements abstract fractional permissions. So, as with predicates, we define an uninterpreted function that represents the fraction of permission that the verifier uses whenever it encounters an abstract fractional read permission in a monitor. To make mixing monitors and predicates easier, we use the same global fraction for monitors as we used for predicates.

The really difficult part of handling monitors, however, is the implementation of the locking and deadlock prevention itself. In this section we present our partial solution in the

hope that it will help in finding a working implementation of deadlock avoidance in SIL, or failing that, in the hope that it at least serves as a demonstration of the limits of SIL.

3.4.1 Approach to Deadlock Prevention and Locking

For locking and deadlock prevention we need to add two kinds of information to each object: a boolean indicating whether the object in question is locked and a *mu* value that indicates the object's position in the locking order. To recapitulate, in section 2.1.7 we saw that mu values are part of the partially ordered set $(Mu, \ll)$. Only one concrete value exists: *lockbottom*, the single smallest element of Mu. All other values of Mu are kept abstract and only described in terms of their $\ll$-relation to one another.

The key idea behind deadlock prevention in Chalice, that a thread can only acquire a lock on monitor/object m if it is clear that that monitor/object is higher in the locking order than any other monitor that that thread already has a lock on. In more concrete terms, locking is allowed when $\forall o \in \text{objects}.\ o \neq m \Rightarrow (\texttt{holds}(o) \Rightarrow o.\texttt{mu} \ll m.mu)$.

To use this expression in a SIL assertion (an exhale statement prior to acquiring a lock), we need to make sure that the heap locations it mentions ($o.\texttt{mu}$ and $m.\texttt{mu}$) are defined: that these locations are readable. To ensure that we can access $m.\texttt{mu}$, we first check $m \neq \texttt{null} \wedge 0 < \texttt{perm}(m.\texttt{mu})$. But then we run into the problem with $o.\texttt{mu}$. Since o is quantified over *all* object references, we would have to require read access to all `mu` fields in the entire heap to ensure that $o.\texttt{mu}$ is defined *for all* object references. Alternatively, one could try to use $0 < \texttt{perm}(o.\texttt{mu})$ to make sure that $o.\texttt{mu}$ is only accessed when we know that the location is readable:

$$0 < \texttt{perm}(o.\texttt{mu}) \Rightarrow \neg\texttt{holds}(o) \vee o.\texttt{mu} \ll m.mu$$

Unfortunately that is not correct. While a method needs read-access to the `mu` field to acquire the lock on an object o_1, it can then give away all permissions to fields of o_1 (it could fork a thread and hand complete control over that object to the forked thread), while still holding the lock on the monitor associated with o_1. With the implication in the assertion above, we are really *filtering* out object references for which we do not have access to the `mu` field. Temporarily not having any permission to access to $o_1.\texttt{mu}$ would allow us to acquire the lock on a monitor that is not higher in the locking order than o_1. Instead our assertion should be quantified over *all* object references o, whether we have access to $o.\texttt{mu}$ or not.

We need to associate each object with a `mu` and a flag that indicates whether the object's monitor is acquired by the current thread. The only mechanism that SIL provides for associating information with an object is fields, but using fields is not an option when one needs to refer to that field in a universal quantifier. The Boogie-based implementation uses both `mu` fields *and* a set of additional masks and that is what we will try to emulate in our approach.

We add another hidden parameter to each method, the `currentThread: ref`. This object has two fields `heldMap` and `muMap`, mapping from references to boolean values and from references to mu values, respectively. For any object reference o, we use $\texttt{heldMap}[o]$ to indicate whether the object o is locked by the current thread at the moment. We use $\texttt{muMap}[o]$ instead of $o.\texttt{mu}$ inside quantifiers to get around the issue of $o.\texttt{mu}$ not being readable. Via

their preconditions, each method gets to assume the following:

$$\texttt{currentThread} \neq \texttt{null} \wedge \texttt{acc}(\texttt{currentThread.heldMap}, \texttt{write}) \wedge \texttt{acc}(\texttt{currentThread.muMap}, \texttt{write})$$

However, the `mu` field is not an implementation detail of Chalice2SIL or the Boogie-based Chalice verifier, but an actual field that the user of Chalice has to take into consideration when dealing with monitors. The greatest challenge is thus to keep what the verifier knows about the `muMap` and what it knows about `mu` fields in synchrony. To that end, every occurrence of an accessibility predicate that mentions `mu` will be translated according to the following rule:

$$[\![\texttt{acc}(x.\texttt{mu}, a)]\!]_{\text{Ch}} = [\![\texttt{acc}(x.\texttt{mu}, a) \wedge \texttt{currentThread.muMap}[x] = x.\texttt{mu}]\!]_{\text{SIL}}$$

This makes sure that whenever we gain access to an `mu` field, we also get a matching entry in the `currentThread`'s `muMap`. Or, conversely, whenever we give away permission to a `mu` field, we must also ensure that the `muMap` is up to date.

With this infrastructure, the implementation of locking-related operations is relatively straightforward. For the examples that follow, we assume that we have a class C with a single `int` field f.

Object creation $[\![o$ `:= new` $C]\!]_{\text{Ch}}$ is translated as

```
o := new ref;
inhale acc(o.f,write); // for each field f of class C
inhale acc(o.mu,write);
inhale o.mu == lockbottom;
inhale currentThread.heldMap[o] == false;
inhale currentThread.muMap[o] == o.mu;
```

Share an object $[\![$`share` o `above` a `below` $b]\!]_{\text{Ch}}$ is translated as

```
exhale o != null;
exhale o.mu == lockbottom;
// Ensure bounds are defined
exhale a != null && b != null;
exhale 0 < perm(a.mu) && 0 < perm(b.mu);
exhale a << b; // Upper bound might be below lower bound
// Constrain value for fresh mu
var m : Mu;
inhale lockbottom << m;
inhale a << m;
inhale m << b;
// Assign mu (to both the field and the map), set held to false
o.mu := m;
currentThread.muMap[o] := o.mu;
currentThread.heldMap[o] := false;
exhale <monitor-invariant>;
```

[LMS09] also introduces other forms of Chalice's `share` statement allow the programmer to specify multiple upper and lower bounds or omit them altogether. If no

lower bound is specified, the current *lock level* is used as the lower bound, that is, the object is shared with a `mu` that is greater than the `mu` of any object for which the thread currently holds a lock. After an object has been shared, its `mu` field is guaranteed to be above `lockbottom`. This property is used to determine whether an object is currently shared and can often be found in the precondition of methods that intend to lock that object.

Acquire a lock $[\![\texttt{acquire}\ o]\!]_{\text{Ch}}$ is translated as

```
exhale o != null;
exhale 0 < perm(o.mu);
// o must have been shared above current waitlevel
exhale forall h : ref :: currentThread.heldMap[h] ==>
                                        currentThread.muMap[h] << o.mu;
currentThread.heldMap[o] := true;
inhale <monitor-invariant>;
```

Release a lock $[\![\texttt{release}\ o]\!]_{\text{Ch}}$ is translated as

```
exhale o != null;
exhale currentThread.heldMap[o];
exhale <monitor-invariant>;
currentThread.heldMap[o] := false;
```

Unshare an object $[\![\texttt{unshare}\ o]\!]_{\text{Ch}}$ is translated as

```
exhale o != null;
exhale write <= perm(o.mu);
exhale lockbottom << o.mu; // ensures o is shared
exhale currentThread.heldMap[o]; // o is locked
// Update fields/maps
o.mu := lockbottom;
currentThread.heldMap[o] := false;
currentThread.muMap[o] = o.mu;
```

Forking a thread Chalice2SIL creates a new thread object for the forked thread and initialises its `muMap` with the contents with the contents of `currentThread`'s `muMap`. Since the `heldMap` only represents the locks held by the *current thread* we do not copy anything from `currentThread`'s `heldMap`.

In the postcondition of each method, we make sure that the `mu` and `held` maps are in a consistent state. The programmer uses the **`lockchange`** declaration to list all objects it changed the lock state of. Without any **`lockchange`** declarations, all method postcondition contain the following assertions:

$$
\begin{aligned}
&\texttt{acc}\,(\texttt{currentThread.heldMap}, \texttt{write}) \wedge \texttt{acc}\,(\texttt{currentThread.muMap}, \texttt{write}) \\
&\wedge\ \forall o_c \in \text{objects.}\ \texttt{old}\,(\texttt{currentThread.heldMap})\,[o_c] = \texttt{currentThread.heldMap}[o_c] \\
&\wedge\ \forall o_d \in \text{objects.}\ (\texttt{currentThread.heldMap}[o_d]) \Rightarrow \\
&\qquad \texttt{old}\,(\texttt{currentThread.muMap}[o_d]) = \texttt{currentThread.muMap}[o_d]
\end{aligned}
$$

The second line demands that the locking state of any object must not have been changed. The last two lines ensure that the lock level of anything that is currently locked didn't

Listing 18: Losing information about mu.

```
1  class C {
2      var f : int;
3      invariant acc(f);
4      method nop(){}
5      method main()
6      {
7          var x := new C; var y := new C;
8          share x;
9          share y above x;
10
11         call nop();
12         acquire x;
13         acquire y;      release y;
14         release x;
15     }
16 } // Using Syxc: Verification finished with 0 error(s)
```

change. When the user adds **lockchange** declarations, the bodies of the last two quantifiers are guarded by an implication similar to:

$$o \notin \texttt{lockchange} \Rightarrow \text{body...}$$

3.4.2 Limitations of the current Implementation

As mentioned above, the current implementation is not correct in some cases. We identified two major problems: one where we lose information about mu, preventing us from successfully verifying a correct program, and one where we retain too much information about mu, causing the verification to be potentially unsound.

Listing 18 presents an example of the former problem. In that program we create two objects x and y and share them in such a way that locks on x always have to be acquired before y (lines 8 and 9). When we call the method nop on line 11, we temporarily give away all permissions to currentThread.muMap. This means that the verifier must assume that the contents of the heap location currentThread.muMap have changed completely. When the verifier reaches the second **acquire** statement on line 13, it will assert the following:

```
forall h : ref :: currentThread.heldMap[h] ==>
                                        currentThread.muMap[h] << y.mu;
```

Since there is only one locked object at the moment, the verifier effectively checks

```
currentThread.muMap[x] << y.mu
```

It remembers that x.mu << y.mu, but has lost all information about muMap[x]. As a result, the assertion will fail.

Listing 19: Modified method nop from listing 18, causes verification to succeed.

```
method nop(x : C)
    requires acc(x.mu,rd)
    ensures  acc(x.mu,rd)
{ }
```

Listing 20: Keeping too much information about mu.

```
1  class C {
2      var f : int;
3      invariant acc(f);
4      method unrelated(x : C)
5          requires x != null && acc(x.mu)
6          ensures acc(x.mu)
7      { }
8
9      method main()
10     {
11         var x := new C; var y := new C;
12         share x;
13         share y above x;
14         acquire x;
15         fork unrelated(x);
16         acquire y;     release y;
17         release x;
18     }
19 }// Using Syxc: Error 1280: 17.17: Acquiring y failed. waitlevel << mu
       might not hold.
```

We can work around this limitation by explicitly mentioning `x.mu` in an accessibility predicate on the pre- and postcondition of method `nop`. That causes the assertion `x.mu == currentThread.muMap[x]` to be included in the pre- and postcondition of the translated SIL method.

In listing 20 we first acquire the lock on `x` and then fork a new thread, giving away write permission to `x.mu`. When we try to `acquire y` on line 17, the verifier tries to assert the same expression as in the last example, only here this causes an outdated value for `currentThread.muMap[x]` to be used. Since all permissions to the heap location `x.mu` have been given away as part of the **`fork`** statement in line 16, `x.mu` might no longer have the same value.

Both issues are related to the problem of finding a method's *frame*, figuring out which locations a method cannot access or modify. In the first case, we are missing the fact that the method `nop` *cannot change* `x.mu`, whereas in the second case we ignore the fact that the method `unrelated` could potentially have changed `x.mu` by the time we arrive at the **`acquire`** `y` statement.

4 Evaluation

4.1 SIL as a translation target/verification intermediate language

One of the primary goals of writing Chalice2SIL was to gather experience working with SIL, both as a translation target and as a verification intermediate language.

4.1.1 Encoding of loops

At the time we started this project, SIL did not have a dedicated `while` loop node. Instead, programs were to be encoded as a flat directed graph of basic blocks: a control-flow graph (CFG). Loops were encoded as cycles with the "backwards" pointing edge explicitly marked (so that tools could traverse the CFG as an acyclic graph by ignoring those back edges). Unfortunately, Silicon – our verifier for SIL – can currently only handle `while` loops as they appear in Chalice. In those early days, Silicon would pattern match against the CFG to find `while` loops and extract their components (condition, invariant, body), essentially lifting the program back up to the abstraction level of Chalice in terms of control flow. If Chalice, which only supports `while` loops, were the only source language that SIL ever had to support, this approach would have been fine. But since the idea behind SIL was to eventually have multiple front ends, we decided to capture the fact that we currently can only verify `while` loops – and not arbitrary control flow graphs – in the language. As a result a explicit loop node was added to the SIL control-flow graph.

4.1.2 Syntactic distinction between assertions and program expressions

Currently, SIL distinguishes between assertions (logical formulae and accessibility predicates) and program expressions on a syntactic level. Some language elements require assertions as operands (e.g., `exhale`) while others only accept program expressions (e.g., method arguments). In the current implementation of the SIL abstract syntax tree (AST), there are two distinct and unrelated types: the type of assertions and the type of program expressions. Having the Scala compiler enforce that we never construct a SIL program where an accessibility predicate is used as a method argument is nice in theory, but proved to be more cumbersome than necessary in practice.

Only a partial solution It is still possible to write translators that try to create illegal assertions and will fail due to runtime[1] checks built into the SIL AST. While it would seem better to check as many properties statically as possible, only ending up with partial checks can result in a false sense of security. The SIL AST API is not exception-free and translators should be prepared deal with exceptions.

Code duplication Logical formulae and program expressions have a lot in common, e.g. logical operators or literal values. Distinguishing between a program-level `true` and and an assertion-level `true` has little benefit and at the same means that both producers and consumers of SIL need to have two pieces of code that handle boolean literals. Since the Scala types of assertions and program expressions are unrelated, there is absolutely no opportunity for code reuse.

[1] In the context of *X*-to-SIL-translators, "runtime" refers to the execution of the translator.

Translation from Chalice The Chalice compiler does not distinguish between assertions and program expressions on a syntactic level and instead enforces restrictions on where certain expressions can appear in semantic checks during type checking. Unfortunately, this makes syntax driven translation from Chalice to SIL highly ambiguous. The translator will come across many Chalice expressions where it is not a priori clear whether to translate them as SIL assertions or as SIL program expressions. For instance $[\![5 == 3]\!]_{\text{Ch}}$ can be translated using the equality assertion $[\![\texttt{5 == 3}]\!]_{\text{SIL}}$ or by first applying the integer equality domain function `intEQ` and then lifting the resulting boolean program value up to assertion-level using the boolean domain predicate `eval`: $[\![\texttt{eval(intEQ(5,3))}]\!]_{\text{SIL}}$.

Not all expressions can be translated either way and to find out which translation scheme is the correct one often requires having a look at the entire expression tree and not just the outermost expression node. This can mean that a translator needs to walk through Chalice expressions twice, either just trying both translation schemes in turn or first analysing the expression and then deciding on a translation scheme to use.

Need to convert For Chalice2SIL it was sometimes necessary to convert between assertions and program expressions. One example of this are expressions of the form $[\![\texttt{old}(e)]\!]_{\text{Ch}}$ where e translates to a SIL assertion. When a method with such an `old` expression in its precondition is forked, conceptually, the expression e is evaluated and its "value" (`true` or `false`) stored in a field on the fork-token. Because the right-hand side of an assignment needs to be a program expression in SIL, we first have to create a fresh boolean variable and associate the truth value of that variable with the assertion:

```
var b : bool;
inhale eval(b) == e
```

This can always be done and thus making the translator (and later the verifier) jump through these hoops seems pointless.

Naming In the actual implementation of the SIL AST, assertions are called *expressions* and program expressions are called *terms*. While the implementation uses this terminology very consistently, the terms fail to convey the key difference between assertions and program expressions, resulting in a lot of puzzled faces in conversations with people who are not intimately familiar with SIL's design.

4.1.3 PTerms vs. DTerms vs. GTerms vs. Terms

SIL makes another distinction on the syntactic level that we think is better handled as a semantic check. To make sure that domain axioms don't contain AST nodes that are illegal in the context of a domain (such as references to heap locations), SIL has four types to represent program expressions.

DTerm "Domain" terms represent the set of all program expressions that are legal in domain axiom specifications. References to quantifier variables are one example.

PTerm "Program" terms represent the set of all program expressions that are legal in actual program code (such as the right-hand side of assignment statements). Heap references are one example.

GTerm "General" terms represent the set of all program expressions that are legal in *all* contexts. Integer literals are one example.

Term Represent the set of all program expressions. Examples include the full permission amount `write` or the permission mask lookup **`perm`** $(x.f)$.

Even though the SIL AST implementation uses Scala traits to capture the subset relationships between these sets, there is still an enormous amount of code duplication. For instance, it is not enough to have one node type for domain function applications. Because you need to restrict the set from which the function application node draws its arguments, there is one domain function application node type for each of the four sets: `GDomainFunctionApplication`, `DDomainFunctionApplication`, `PDomainFunctionApplication` and `DomainFunctionApplication`.

Luckily, the subset types are all subtypes of `DomainFunctionApplication` which allows for code reuse when *consuming* (pattern matching) these data structures. When it comes to *generating* these nodes, however, we essentially had two options. One option was to duplicate a lot of code (e.g., have separate translation schemes for `DDomainFunctionApplication` and `PDomainFunctionApplication`). We chose instead to translate to the most specific type whenever possible (e.g., if all arguments of a domain function application are `DTerms` themselves, create a `DDomainFunctionApplication`) but return `Terms` to accommodate for all Chalice program expressions. In cases where `PTerms` are required instead of `Terms`, we attempt to downcast to `PTerm`.

The fact that even the SIL AST implementation itself employs this technique indicates that maybe in this case runtime checks (running during translation of a program to SIL) are better suited than trying to encode these constraints in the Scala type system.

The SIL AST implementation makes a similar distinction for assertions. There are `GExpressions`, `DExpressions`, `PExpressions` and `Expressions`. We included a more complete grammar listing in appendix A.

4.1.4 Capturing state in SIL

A pattern that often appears in the Boogie-based implementation of Chalice, is that one would make a copy of the heap and permission mask (both are ordinary variables from Boogie's perspective) then perform a series of operations and assertions on that copy (e.g., inhale, exhale). Describing programs on a higher level of abstraction, SIL does not allow anything similar. The `perm` expression is about as close as we get to the permission mask.

Re-implementing the permission mask as we did to constrain the read fractions (section 3.1.1) seems incredibly wasteful since most tools that work on SIL will have a concept of a permission map, maybe even specialised code to deal with them and all they see are manipulations of an abstract data structure (the map created by the Chalice2SIL translation), described by a couple of axioms. So far this hasn't been a serious problem, though.

A similar issue is how we currently handle **`old`** expression for fork/join. What we would ideally like to do is to capture the program state at the point where a thread is forked off and then associate that state with the token. Later when we **`join`** on the token, we just evaluate the **`old`** expression in the state associate with the token. We would not have to worry about the defined-ness of **`old`** expressions at the point where we fork the token.

Listing 21: Error that is not detected by Chalice2SIL+Silicon.

```
class C {
    var x : int;
    predicate V { acc(x) }

  function failUnfoldingV(): int
  { unfolding V in x }   // should not be able to unfold V, but succeeds
}
```

Listing 22: SIL translation of 21

```
field C::x : Integer
function C::failUnfoldingV() : result
  = unfolding acc(this.C::V,write) in (this.C::x)
predicate C::V = acc(this.C::x,write)
```

Chalice has other features in which old state is referenced. History constraints on monitor invariants and general **eval** expressions are examples.

4.2 Chalice2SIL+Silicon compared to Syxc

To evaluate our implementation we compared it to "*Syxc*", another Chalice verifier and the tool from which Silicon was derived. Both Chalice2SIL+Silicon and Syxc use the parser and type checker from the original Chalice implementation and perform the actual verification using a combination of symbolic execution and calls to the Z3 SMT solver.

We took a large portion of Syxc's test suite and compared the results of passing those tests to Chalice2SIL+Silicon and Syxc, both in terms of correctness and as an ad-hoc performance benchmark. For two results to be considered equal, we required that the tools emitted the same number of error messages on the same lines.

4.2.1 Benchmark: correctness

In total, we looked at 84 test programs (one program usually consisted of multiple test cases) from the Syxc test suite. Files that the current version Syxc[2] itself could not handle were not considered. Out of these 84 files, Chalice2SIL+Silicon were able to correctly verify 45 files without any modifications and additional 5 files with slight modifications. Another 6 files were only verified correctly in parts. In 5 of those cases Chalice2SIL+Silicon reported errors not also reported by Syxc. At least two of these failures as caused by Silicon not merging permissions or being able to unfold a predicate nested in another predicate. For the other case Chalice2SIL+Silicon misses an illegal unfolding in a function (example in listing 21). This is likely a bug in Silicon as the translation to SIL is very straightforward and is included in listing 22.

We are left with 28 files for which Chalice2SIL or Silicon couldn't handle due to missing features or bugs in the tools themselves. Chalice2SIL fails 9 cases with an unexpected

[2]Revision `2bdf7ee59971e2734ccb0aaa2f523786bb1d5e74`

exception and another 10 due to features that we decided not to implement as part of this project.

In 4 out of those 9 cases an error during field name resolution is responsible for Chalice2SIL crashing. The remaining crashes are due to an exception during the translation of an expression that involves `waitlevel`, an exception during the translation of a type expression or an exception during a substitution of program variables in an expression. These are likely simple programming errors that should be easy to fix.

Features not implemented as part of this project in Chalice2SIL+Silicon include

- Sequences
- Channels
- General **`eval`** expressions
- History constraints (**`old`** expressions in monitor invariants)
- Counting permissions (implemented in Chalice2SIL, but not in Silicon)

The detailed results of this evaluation have been forwarded to the people who are currently or will be maintaining Chalice2SIL and Silicon.

4.2.2 Benchmark: performance

In addition to comparing the answers of Chalice2SIL+Silicon and Syxc, we also measured how long parsing, translation and verification of each test case took. This performance benchmark is mainly intended to get a rough idea of how high the price of translating to an intermediate language (SIL) is.

For Chalice2SIL+Silicon we measured the time spent in the Chalice parser/type checker, Chalice2SIL itself, and Silicon (including Z3) separately whereas for Syxc we only obtained one measurement per run (excluding JVM start up and argument parsing). For the benchmarks we used a machine with an Intel Core i7 2700K @ 3.9GHZ, 16GiB of DDR-1600 RAM and a mechanical hard drive. Each test was measured three times to compensate for measurement errors. The averaged data is included in tables 1 and 2 in appendix B.

Figures 6, 7 and 8 break down the time taken by the Chalice parser/typechecker, Silicon and Chalice2SIL for each test case. Constructing the Chalice AST took longer than the translation to SIL in all but two cases: `ForkJoin/predicates_fork_join` and `PermissionModel/basic`. We omitted the latter from the diagrams for better readability as it is a bit of an outlier with a Silicon runtime of over 4.5s (Syxc only needed 1.2s). It is not clear why verification of this test case via Chalice2SIL+Silicon is that slow.

As the runtimes of the tests in the benchmark sometimes vary greatly between different test cases, looking at absolute times is not a meaningful way to compare Chalice2SIL and Syxc. Instead we look at the factor by which Chalice2SIL+Silicon is slower than Syxc. In the majority of cases this factor is between 1.6 and 1.8, but there are also cases where Chalice2SIL is 2.5+ times slower. Figure 9 shows how factors were distributed in this benchmark. Two outliers with ratios 3.69 and 5.02 were omitted from this diagram in the interest of readability.

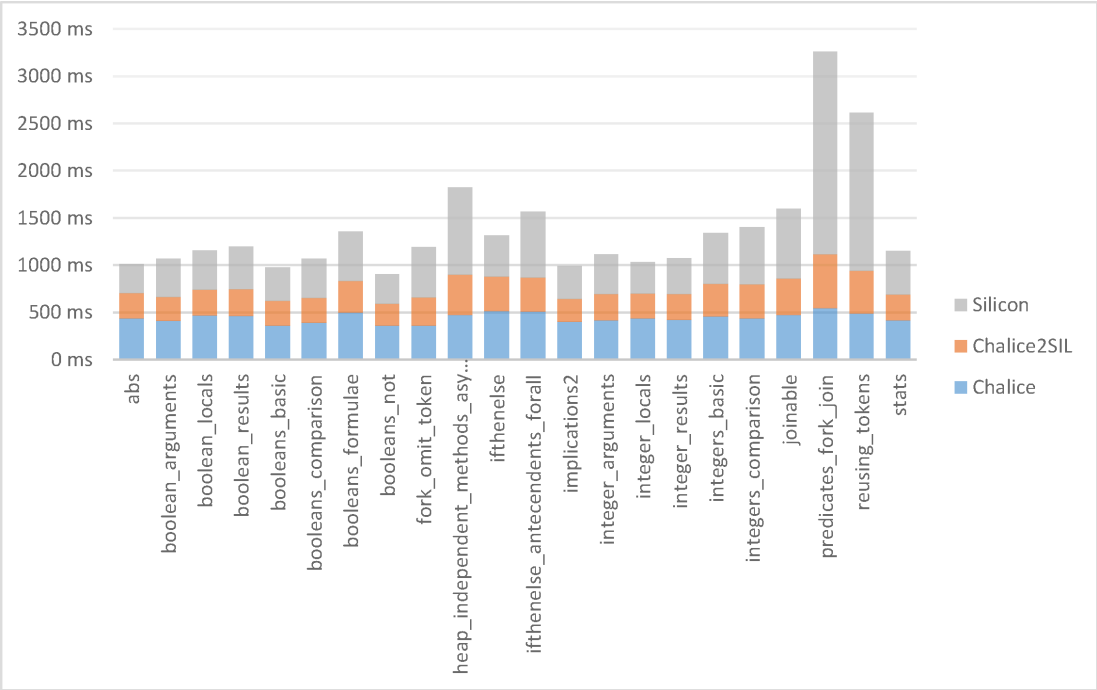

Figure 6: Running Chalice2SIL+Silicon for tests in `Basics/`, `Branching/` and `ForkJoin/`

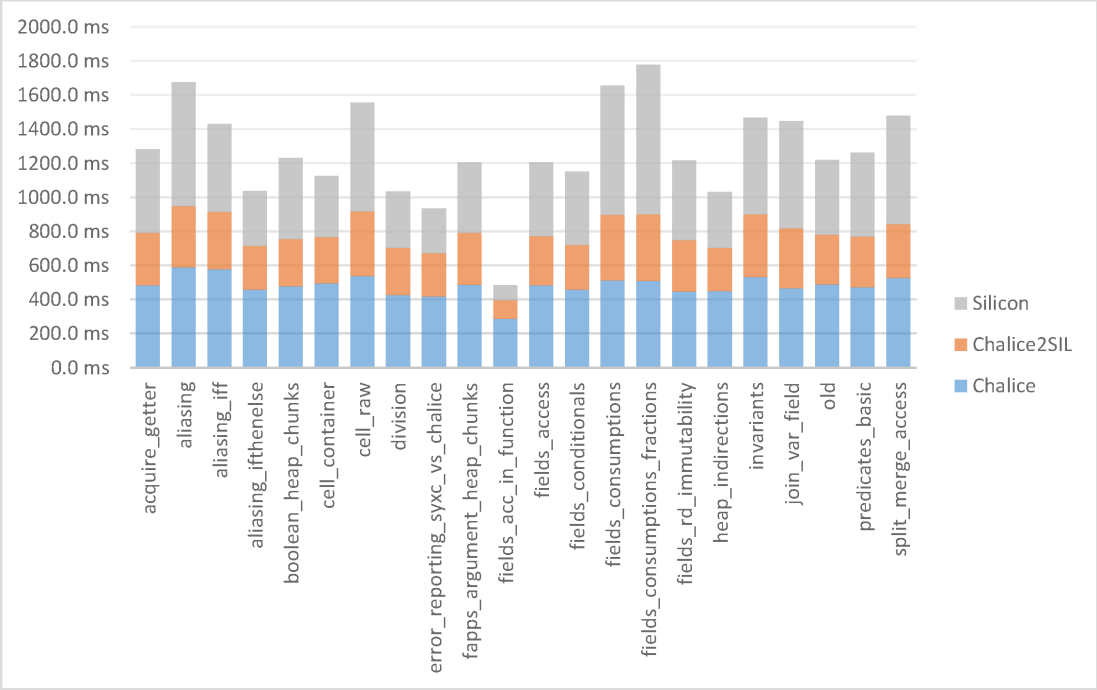

Figure 7: Running Chalice2SIL+Silicon for tests in `Heaps/`, `Misc/` and `Monitors/`

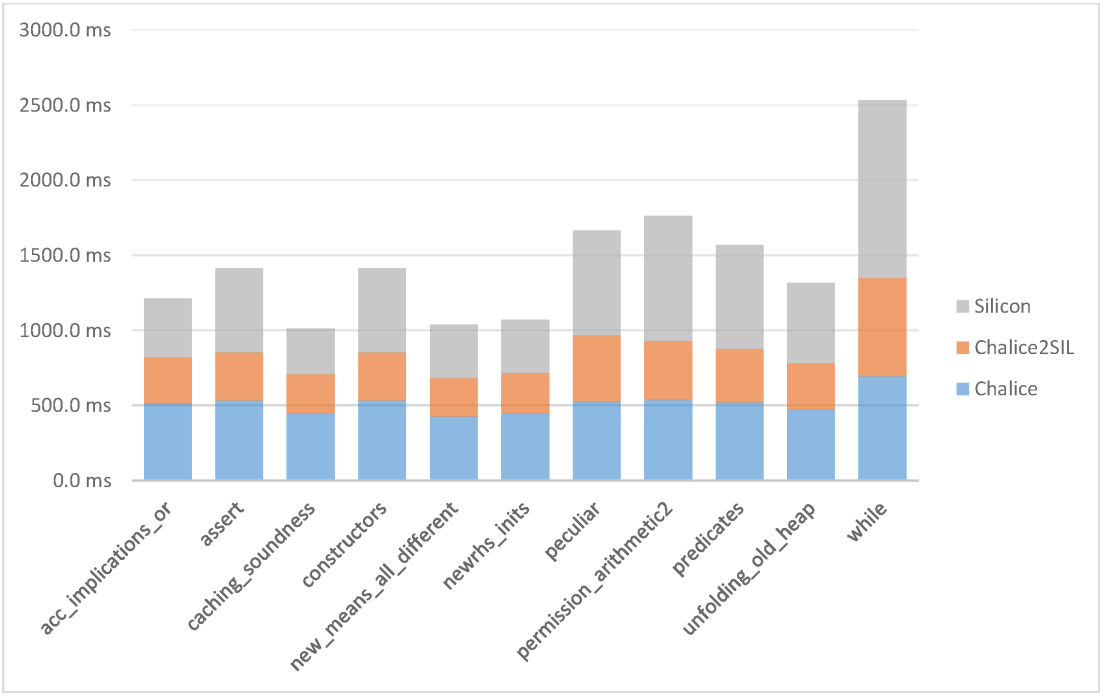

Figure 8: Running Chalice2SIL+Silicon for tests in `PermissionModel/`, `VariousFeatures/`

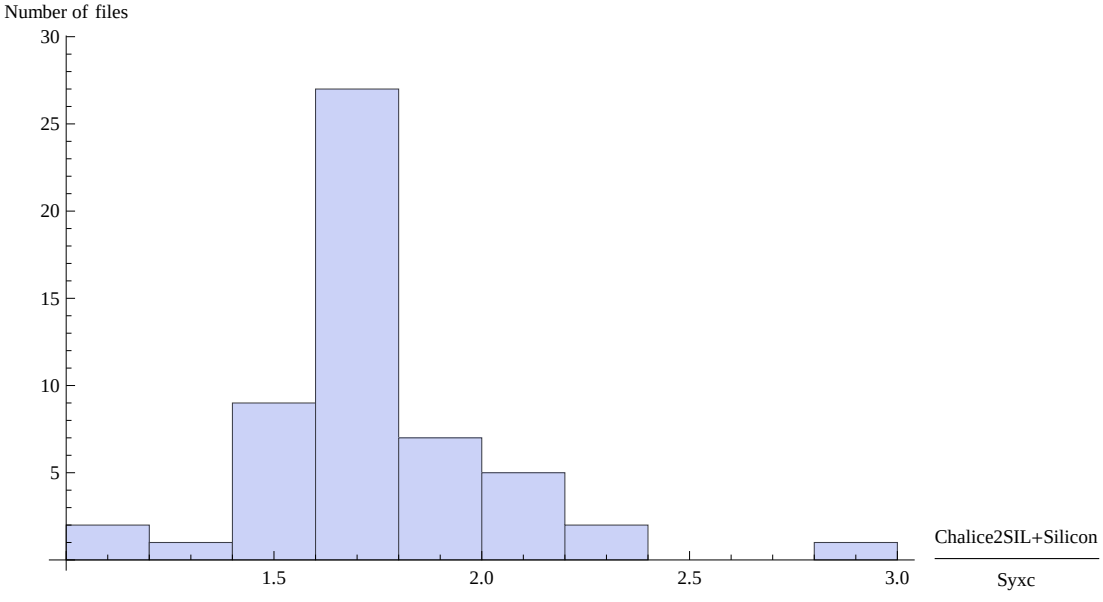

Figure 9: Distribution of the factor by which Chalice2SIL+Silicon is slower than Syxc

4.3 Implementation status

Chalice2SIL implements most of Chalice's core functionality and uses nearly all of the features that SIL has to offer. However, in addition to monitors and deadlock avoidance there are a number of gaps that still need to be filled to make Chalice2SIL+Silicon a true alternative to Syxc or the original Boogie-based Chalice verifier.

In addition to the tests run as part of the evaluation, we also wrote a set of 94 test programs along with expected results that served as part of an automated test suite during development. Overall, Chalice2SIL should be considered a prototype with room for improvement, especially in terms of performance.

5 Conclusion

As part of this project, we devised and implemented a translator from Chalice ASTs to SIL ASTs from scratch. When the project started, not a single line of code existed for the SIL AST, Silicon or Chalice2SIL. SIL itself was little more than a draft of its syntax on paper. This turned out to be both a blessing and a curse. On the one hand just about everyone we talked to had a slightly different idea of how a particular SIL construct was supposed to behave, at least initially. On the other hand we had the opportunity to help shape SIL and the design of its AST.

While many parts of the translation from Chalice to SIL were comparatively straightforward due to the similarity between the two languages, some aspects of Chalice or its underlying permission model were surprisingly hard to implement within the constraints of SIL. Overall, however, we think that the high-level design of SIL goes in the right direction. Having permission amounts as first class values and related constructs (`acc`, `exhale`, etc.) as built-in language constructs neatly decouples the representation of permissions in the verifier from the representation of the program to be verified.

With Chalice2SIL+Silicon we now have a first prototype of an automatic program verification tool-chain based on SIL, ready to be extended to include other tools that consume or transform SIL programs.

A Full SIL Term and Expression Grammar

⟨*Expr*⟩ ::= 'acc' (⟨*Location*⟩, ⟨*Term*⟩)
| 'old' (⟨*Expr*⟩)
| 'unfolding' ⟨*Term*⟩.⟨*pred-id*⟩ 'by' ⟨*Term*⟩ 'in' ⟨*Expr*⟩
| ⟨*Term*⟩ == ⟨*Term*⟩
| ⟨*unary-op*⟩ ⟨*Expr*⟩
| ⟨*binary-op*⟩ ⟨*Expr*⟩
| ⟨*dom-pred-id*⟩({ ⟨*Term*⟩ , ⋯ })
| '∀' ⟨*logical-var-id*⟩ : ⟨*DataType*⟩ :: (⟨*Expr*⟩)
| '∃' ⟨*logical-var-id*⟩ : ⟨*DataType*⟩ :: (⟨*Expr*⟩)
| ⟨*GExpr*⟩

⟨*Location*⟩ ::= ⟨*Term*⟩.⟨*field-id*⟩
| ⟨*Term*⟩.⟨*pred-id*⟩

⟨*PExpr*⟩ ::= 'acc' (⟨*PLocation*⟩, ⟨*PTerm*⟩)
| 'unfolding' ⟨*PTerm*⟩.⟨*pred-id*⟩ 'by' ⟨*PTerm*⟩ 'in' ⟨*PExpr*⟩
| ⟨*PTerm*⟩ == ⟨*PTerm*⟩
| ⟨*unary-op*⟩ ⟨*PExpr*⟩
| ⟨*binary-op*⟩ ⟨*PExpr*⟩
| ⟨*dom-pred-id*⟩({ ⟨*PTerm*⟩ , ⋯ })
| ⟨*GExpr*⟩

⟨*PLocation*⟩ ::= ⟨*PTerm*⟩.⟨*field-id*⟩
| ⟨*PTerm*⟩.⟨*pred-id*⟩

⟨*DExpr*⟩ ::= ⟨*DTerm*⟩ == ⟨*DTerm*⟩
| ⟨*unary-op*⟩ ⟨*DExpr*⟩
| ⟨*binary-op*⟩ ⟨*DExpr*⟩
| ⟨*dom-pred-id*⟩({ ⟨*DTerm*⟩ , ⋯ })
| '∀' ⟨*logical-var-id*⟩ : ⟨*DataType*⟩ :: (⟨*DExpr*⟩)
| '∃' ⟨*logical-var-id*⟩ : ⟨*DataType*⟩ :: (⟨*DExpr*⟩)
| ⟨*GExpr*⟩

⟨*GExpr*⟩ ::= ⟨*PExpr*⟩ == ⟨*PExpr*⟩
| ⟨*UnaryOp*⟩ ⟨*PExpr*⟩
| ⟨*BinaryOp*⟩ ⟨*PExpr*⟩
| ⟨*dom-pred-id*⟩({ ⟨*PExpr*⟩ , ⋯ })
| 'True'
| 'False'

⟨*UnaryOp*⟩ ::= '¬'

⟨*BinaryOp*⟩ ::= '∧' | '∨' | '≡' | '⇒'

⟨*Term*⟩ ::= 'if' ⟨*Term*⟩ 'then' ⟨*Term*⟩ 'else' ⟨*Term*⟩
| 'old'(⟨*Term*⟩)
| ⟨*func-id*⟩({ ⟨*Term*⟩ , ⋯ })
| ⟨*dom-func-id*⟩({ ⟨*Term*⟩ , ⋯ })
| 'unfolding' ⟨*Term*⟩.⟨*pred-id*⟩ 'by' ⟨*Term*⟩ 'in' ⟨*Term*⟩
| (⟨*Term*⟩) : ⟨*DataType*⟩
| ⟨*Term*⟩.⟨*field-id*⟩
| 'perm'(⟨*Location*⟩)
| 'write'
| '0'
| 'E'
| ⟨*GTerm*⟩

SIL has three literal constants for permission amounts: `write` denotes the full permission, `0` stands for no permission at all (it is distinct from the integer 0 literal) and `E` represents a single epsilon of permission (a counting permission, see section 2.1.3).

⟨*PTerm*⟩ ::= 'if' ⟨*PTerm*⟩ 'then' ⟨*PTerm*⟩ 'else' ⟨*PTerm*⟩
| ⟨*var-id*⟩
| ⟨*func-id*⟩({ ⟨*PTerm*⟩ , ⋯ }
| ⟨*dom-func-id*⟩({ ⟨*PTerm*⟩ , ⋯ })
| 'unfolding' ⟨*PTerm*⟩.⟨*pred-id*⟩ 'by' ⟨*PTerm*⟩ 'in' ⟨*PTerm*⟩
| (⟨*PTerm*⟩) : ⟨*DataType*⟩
| ⟨*PTerm*⟩.⟨*field-id*⟩
| ⟨*GTerm*⟩

⟨*DTerm*⟩ ::= 'if' ⟨*DTerm*⟩ 'then' ⟨*DTerm*⟩ 'else' ⟨*DTerm*⟩
| ⟨*logical-var-id*⟩
| ⟨*dom-func-id*⟩({ ⟨*DTerm*⟩ , ⋯ })
| ⟨*GTerm*⟩

⟨*GTerm*⟩ ::= 'if' ⟨*GTerm*⟩ 'then' ⟨*GTerm*⟩ 'else' ⟨*GTerm*⟩
| ⟨*integer-literal*⟩
| ⟨*dom-func-id*⟩({ ⟨*GTerm*⟩ , ⋯ })

B Benchmark data

Table 1: Benchmark data, part 1

Group	Name	Chalice	ToSil	Silicon	Total	StdDev	Syxc	StdDev
Basics	boolean_arguments	409.7	250.0	407.7	1067.3	6.7	650.7	17.6
Basics	boolean_locals	465.3	274.3	414.3	1154.0	5.2	707.0	13.0
Basics	boolean_results	464.3	279.3	454.0	1197.7	25.1	713.3	17.9
Basics	booleans_basic	359.3	260.3	354.0	973.7	9.0	547.0	0.0
Basics	booleans_comparison	390.3	260.3	416.7	1067.3	10.1	609.7	15.5
Basics	booleans_formulae	497.7	333.3	526.0	1357.0	19.1	849.0	4.4
Basics	booleans_not	359.3	229.0	317.7	906.0	1.0	552.0	9.5
Basics	integer_arguments	416.7	275.7	422.3	1114.7	9.0	677.3	9.2
Basics	integer_locals	437.3	258.0	334.0	1029.3	15.7	614.7	9.0
Basics	integer_results	422.0	270.3	380.0	1072.3	9.0	630.3	9.2
Basics	integers_basic	455.7	344.0	541.3	1341.0	3.5	760.3	17.9
Basics	integers_comparison	437.3	358.0	604.0	1399.3	29.1	734.3	16.0
Branching	abs	432.0	268.3	312.3	1012.7	11.4	591.3	2.5
Branching	ifthenelse	516.0	359.3	437.7	1313.0	1.0	864.7	24.0
Branching	ifthenelse_antecendents_forall	510.0	354.0	703.0	1567.0	9.5	906.3	0.6
Branching	implications2	401.0	240.3	349.3	990.7	18.8	584.0	10.6
Branching	stats	412.3	273.7	460.0	1146.0	13.2	573.0	9.5
ForkJoin	fork_omit_token	359.7	297.0	536.7	1193.3	9.3	546.7	27.1
ForkJoin	heap_independent_methods_async	474.3	425.3	922.0	1821.7	23.0	760.7	23.5
ForkJoin	joinable	474.0	382.7	740.0	1596.7	15.9	674.3	4.9
ForkJoin	predicates_fork_join	541.7	571.3	2145.7	3258.7	29.7	1151.7	17.6
ForkJoin	reusing_tokens	485.7	455.7	1672.0	2613.3	50.2	708.0	8.7
PermissionModel	basic	584.0	692.7	4755.0	6031.7	62.0	1201.0	6.1
PermissionModel	caching_soundness	448.0	260.3	302.3	1010.7	8.4	651.0	8.7
PermissionModel	peculiar	523.7	443.3	698.0	1665.0	67.7	818.0	5.6
PermissionModel	permission_arithmetic2	536.7	390.7	833.3	1760.7	9.3	1229.0	8.7
PermissionModel	predicates	521.0	354.0	692.7	1567.7	23.4	916.3	17.9

Table 2: Benchmark data, part 2

Group	Name	Chalice	ToSil	Silicon	Total	StdDev	Syxc	StdDev
Heaps	aliasing	588.3	359.0	729.3	1676.7	9.0	1479.3	9.0
Heaps	aliasing_iff	579.7	336.0	514.7	1430.3	49.0	1139.0	12.2
Heaps	aliasing_ifthenelse	458.3	255.3	322.7	1036.3	9.2	672.0	16.0
Heaps	boolean_heap_chunks	476.3	276.3	478.3	1231.0	10.1	826.0	11.5
Heaps	fapps_argument_heap_chunks	487.3	303.3	415.3	1206.0	16.1	765.7	0.6
Heaps	fields_acc_in_function	286.7	109.3	88.7	484.7	0.6	484.0	0.0
Heaps	fields_access	484.3	286.7	432.7	1203.7	56.9	708.3	18.5
Heaps	fields_conditionals	457.0	263.7	430.7	1151.3	19.9	621.0	114.3
Heaps	fields_consumptions	511.3	384.7	761.0	1657.0	26.0	864.3	9.2
Heaps	fields_consumptions_fractions	509.3	392.0	875.7	1777.0	25.1	953.7	8.7
Heaps	fields_rd_immutability	448.0	299.7	468.7	1216.3	13.6	672.3	15.5
Heaps	heap_indirections	449.3	255.7	326.0	1031.0	6.1	656.0	6.0
Heaps	predicates_basic	473.7	294.0	494.0	1261.7	10.7	775.7	14.4
Monitors	acquire_getter	484.7	307.3	489.7	1281.7	40.8	719.0	0.0
Monitors	invariants	531.0	370.0	567.3	1468.3	15.5	880.3	49.7
Monitors	old	489.7	291.7	438.0	1219.3	31.0	713.7	23.5
Misc	cell_container	494.7	270.7	359.7	1125.0	15.0	787.0	8.7
Misc	cell_raw	536.7	380.7	636.7	1554.0	13.5	923.0	13.0
Misc	division	425.7	276.0	331.7	1033.3	6.1	630.3	8.5
Misc	error_reporting_syxc_vs_chalice	416.7	255.0	261.7	933.3	46.7	541.7	45.3
Misc	join_var_field	468.7	349.0	630.0	1447.7	8.5	682.7	9.3
Misc	split_merge_access	526.0	317.7	635.3	1479.0	8.7	932.3	9.0
Misc	unfold_fold_unchanged	530.3	323.7	548.0	1402.0	27.2	887.7	2.1
VariousFeatures	acc_implications_or	510.3	307.7	395.7	1213.7	32.3	719.0	0.0
VariousFeatures	assert	533.0	321.3	559.0	1413.3	78.5	774.0	6.1
VariousFeatures	constructors	533.0	321.3	559.0	1413.3	78.5	774.0	6.1
VariousFeatures	newrhs_inits	451.0	262.3	358.3	1071.7	11.6	648.0	10.6
VariousFeatures	new_means_all_different	427.0	256.0	354.7	1037.7	10.7	615.0	8.7
VariousFeatures	unfolding_old_heap	476.0	303.3	536.3	1315.7	11.9	794.3	4.6
VariousFeatures	while	693.0	656.3	1182.0	2531.3	180.1	1227.3	7.6

References

[BDJ+06] M. Barnett, R. DeLine, B. Jacobs, B.-Y. E. Chang, and K. R. 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.

[BLS] M. Barnett, K. R. M. Leino, and W. Schulte. The spec# programming system: An overview. In *2004, Construction and Analysis of Safe, Secure and Interoperable Smart devices, volume 3362 of Lecture Notes in Computer Science*, pages 49–69. Springer.

[Boy03] J. Boyland. Checking interference with fractional permissions. In Radhia Cousot, editor, *Static Analysis*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer Berlin / Heidelberg, 2003. 10.1007/3-540-44898-5_4.

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

[HLMS11] S. Heule, K. R. M. Leino, P. Müller, and A. J. Summers. Fractional permissions without the fractions. *Formal Techniques for Java-like Programs (FTfJP)*, 2011.

[LM09] K. Leino and P. Müller. A basis for verifying multi-threaded programs. *Programming Languages and Systems*, pages 378–393, 2009.

[LMS09] K. R. M Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. *Foundations of Security Analysis and Design V*, pages 195–222, 2009.

[Sca12] The scala programming language. http://www.scala-lang.org/, 2012. [Online; accessed 21-November-2012].

[Sch11] M. Schwerhoff. Symbolic execution for Chalice. Master's thesis, Eidgenössische Technische Hochschule Zürich, 2011, 2011.

[SJP12] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.*, 34(1):2:1–2:58, May 2012.