Semper

Translating Chalice into SIL

Bachelor's Thesis
Christian Klauser
Supervisor: Dr. Alex Summers

Chalice2SIL

Translate from Chalice
to Semper Intermediate Language

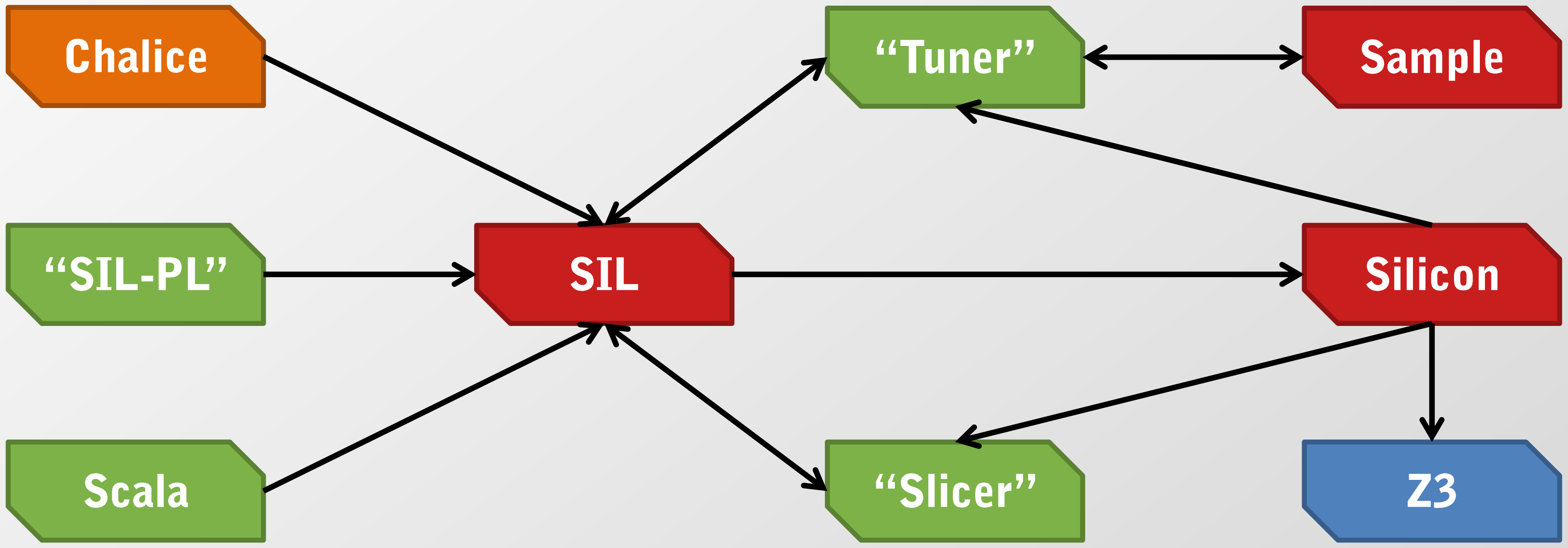
- Chalice
  - verification of concurrent programs
  - uses “permissions” to control access
  - relatively feature-rich
  - major influence on design of SIL

The Semper Project

- Long term project
- Automatic program verifier for Scala
  - verify concurrent programs
  - reduce annotation overhead
  - deal with functional features (e.g., closures)

Scala

Semper Architecture



Semper Intermediate Language (SIL)

- Not a programming language
- A program representation for verification
- Not all constructs are executable
- Aimed at OO
- High-level

```
method C::m(this : ref) : (y:int)
  requires this ≠ null
  requires acc(this.C::f,write)
  ensures acc(this.C::f,write)
  ensures y == this.f

implementation C::m {
  entry:{
    y := this.f;
  }
}
```

SIL Program Structure

- Method Signatures
- Method Bodies
- Domains (data types)
  - predicates
  - functions
  - axioms
- Fields

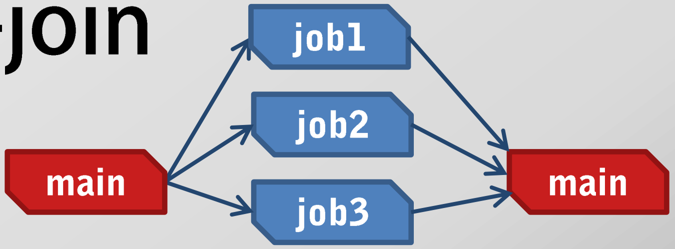
```
method C::m(this :ref)
  requires …
  ensures …

implementation C::m {
  …
}

field C::f : Integer;
```

Permissions

- Tracking
  (Thread × Field × Object) → Permission
- Permission can be

– None	⇒	cannot access at all	“0”
– Some	⇒	can only read	“]0,1[“
– Full	⇒	can read and write	“1”

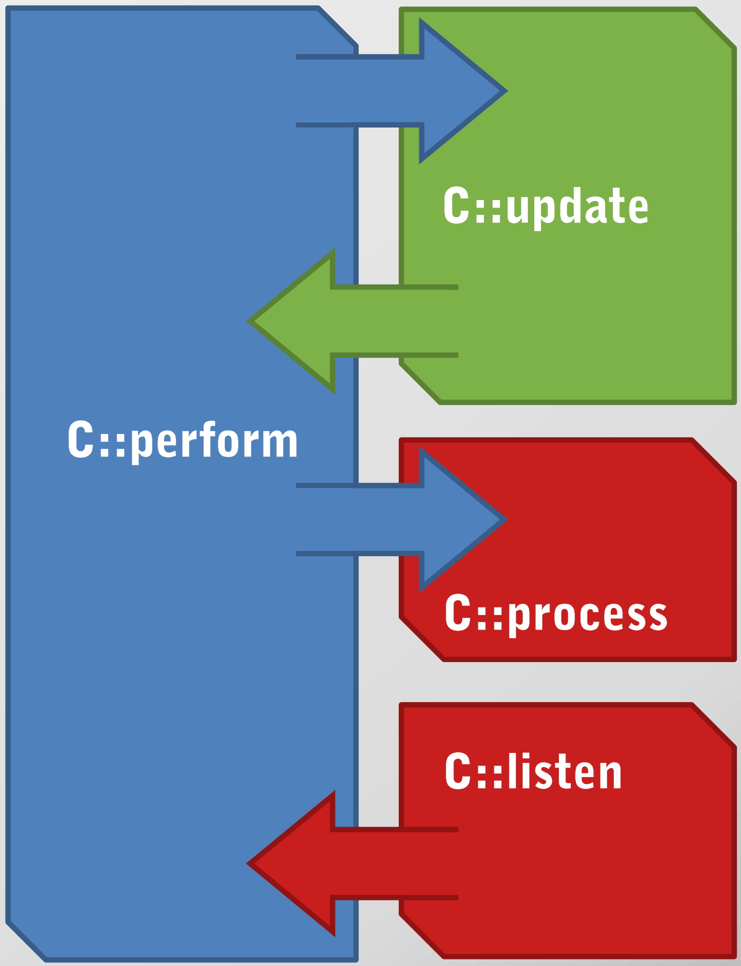
- Neatly supports fork-join



Permission Transfer

Permissions

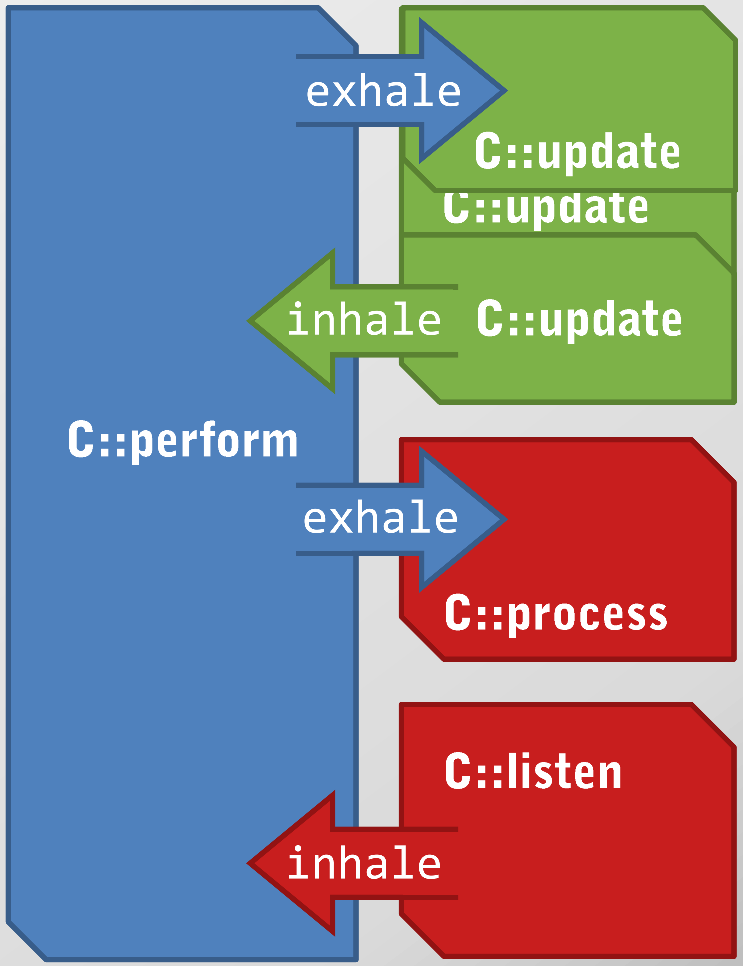
- passed to callee on method entry
- returned to caller on method exit
- passed to other threads on fork
- received from other threads on join



Permission Transfer (Simplification)

Simplification #1:
Method Call
=
Fork + Join

Simplification #2:
Contracts make
transfer unnecessary



Fractional Read Permissions in Chalice

Read-only permissions

- Percent `acc(f,10)`
  - intuitive
  - not composable
  - limited to 100
- Epsilon
  - small, indivisible
  - still not composable

Natively supported in SIL

How does that work?

- Fraction `rd(f)`
  - "$\mathbb{Q} \cap]0,1[$"
  - can always be divided further
  - always abstract
  - concrete value is never computed

Fractional Read Permissions in Chalice

- Fraction `rd(f)`
  - “$\mathbb{Q} \cap]0,1[$”
  - can always be divided further
  - always abstract
  - concrete value is never computed

How does that work?

- Introduce uninitialized permission variable k
- Collect constraints on k as assumptions
- Let abstract fraction be equal to k

Not natively supported in SIL

Implementing Fractional Permissions

- Select one k per method invocation
- Assume that k is a read-permission
- Constrain k for every rd-permission in precondition
- Turn Implications into if-conditions
- Perform call

```
method C::m(this:ref, x:ref,
                        method_k:Permission)
requires 0 < k && k < write
requires acc(this.C::f, method_k)
------------------------------------------
var k : Permission;
inhale 0 < k && k < method_k;

// rd(f) becomes
inhale k < perm(this.f);

// x != null ⇒ rd(x.g) becomes
if(x ≠ null) {
  inhale k < perm(x.g);
}

call C::m(this,x,k);
```

Easy, Right?

Implementing Fractional Permissions #2

- Not correct for `rd(f) && rd(f)`!
- Option 1:
  Don’t use `call`
- Option 2:
  “Simulate” exhale
  without access to permission mask

```
inhale k < perm(this.f);
inhale k < perm(this.f);
```

```
inhale k < perm(this.f);
exhale acc(this.f,k)
```

Implementing Fractional Permissions #3

- Build a map $(\text{ref} \times \text{field}) \rightarrow \text{Permission}$
- Use map for inhale
- Simulate exhale on map

```
inhale k < map[this,f];
map := update(map,(this,f),
                                      map[this,f] - k);
```

Implementing Fractional Permissions #3

- Build a map
  (ref × field) → Permission
  - Add "default value" to map?
  - First build then simulate?
  - Learn about the past!

```
var map : Map[(ref,field),Permission];
var map0 : Map[(ref,field),Permission];
inhale map = map0;
…
if(x ≠ null){
  inhale map0[x,f] = perm(x.f)
  …

}
```

- Use map for inhale
- Simulate exhale on map

```
inhale k < map[this,f];
map := update(map,(this,f),
                             map[this,f] – k);
```

Implement Fork-Join

- No direct equivalent in SIL
- Use `exhale` for `fork`
- And `inhale` for `join`

Not so fast!

- Postcondition might refer to this, args

```
tk := fork m(x);
…
join tk;
```

Solution:

- Store this and args in token
- Link permissions to `.joinable`

Limitations of Fork-Join in Chalice2SIL

Cross-method join

- Don't know what
  `tk.__this`
  means in target method
- Chalice's `eval` expression slated for replacement
- But: Chalice2SIL captures everything it can

Invalid old expressions

- Chalice2SIL captures any
  `old(_)`
  into
  `tk.__old#n`

```
method parallel(d : Dell, b : bool)
    returns (r: bool)
    requires b ==> rd(d.cell)
          && rd(d.cell.f)
    ensures r ==> old(d.cell.f == 5)
          && r == (d != null)
```

QUICK DEMO

State of Chalice2SIL

Implemented

- synchronous calls
- some fork-join scenarios
- predicates
- functions
- testing framework
- good infrastructure

Not Implemented (yet)

- monitors*
  (class invariants, locking)
- deadlock avoidance*
- informative error messages*
- cross-method join
- channels
  (actor-based model)
- stepwise refinement

Conclusion

- SIL and Silicon: a viable backend for Chalice
- SIL conveniently high-level
- Fractional read permissions
  - felt more like fighting SIL
  - direct access to permission mask?
  - permission model plugins?
  - other form of extensibility?

Thank you

QUESTIONS?

BACKUP/SCRAP

Chalice

- Annotated Methods

```
class Cell {
    var v: int;

    method inc(d: int)
        requires 0 < d;
        requires acc(v);
        ensures v == old(v) + d;
    {   v := v + d; }
}
```

Chalice

- Annotated Methods
- Monitors

```
class Cell {
  var v: int;
  invariant acc(v) && 0 <= c;
}
class Program {
  method main() {
    var c:Cell := new Cell;
    c.v := 3;
    share c;

    acquire c; call c.inc(2); release c;
  }
}
```

Chalice

- Annotated Methods
- Monitors
- Predicates/Functions

```
class Cell {
  var v: int;

  predicate valid
  { acc(this.v) && 0 <= this.v }

  function add(d:int) requires valid;
  { unfolding valid in this.v + d; }

  …
}
```

Chalice

- Annotated Methods
- Monitors
- Predicates/Functions
- Fork-Join

```
class Cell { … }
class Program {
  method main()  {
    var c1:Cell := new Cell;
    var c2:Cell := new Cell;
    c1.v := 0; c2.v := 5;
    fork tk1 = c1.inc(3);
    fork tk2 = c2.inc(1);
    join f1 := tk1;
    join f2 := tk2;
  }
}
```

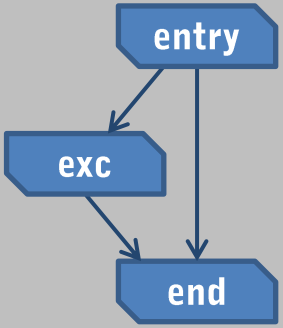
Chalice2SIL

- First front-end for SIL
- Help establish and test the tool chain
- Ideally no changes to Chalice
- If enough time is left
  - Predicates and functions
  - Deadlock avoidance
  - Channels (Actor model)

SIL Program Structure

- Method Signatures
- **Method Bodies**
- Domains (data types)
  - predicates
  - functions
- Fields
- Functions
- Predicates

```
implementation C::cmpexc
{
  entry:{
    o := this.C::f;
  } if(this.C::f = c) goto exc
    if(this.C::f ≠ c) goto end

  exc:{
    this.C::f := v;
  } goto end

  end:{
  }
}
```



$$\begin{tikzcd} & \text{entry} \arrow[dl] \arrow[dd] \\ \text{exc} \arrow[dr] & \\ & \text{end} \end{tikzcd}$$

SIL Program Structure

- Method Signatures
- Method Bodies
- Domains (data types)
  - predicates
  - functions
- Fields
- Functions
- Predicates

```
domain Pair[A,B]{
  function create(A,B)
                          : Pair[A,B];

  function getFirst(Pair[A,B])
                          : A;

  axiom getFirst = ∀ a:A,b:B ::
     getFirst(create(a,b)) = a;
}
domain Permission{
  function
     +(Permission,Permission)
                          : Permission
  predicate
     <(Permission,Permission);
}
```

SIL Program Structure

- Method Signatures
- Method Bodies
- Domains (data types)
  - predicates
  - functions
- Fields
- Functions
- Predicates

```
field C::f : int;
field L::value : int;
field L::next : ref;

function C::fGreater(a : int)
                         : bool
  requires acc(this.C::f,write)
    = C::f>a

predicate L::valid =
  acc(this.L::value,write) &&
  acc(this.L::next,write) &&
  this.L::next≠null ⇒
    acc((this.L::next).L::inv,
                         write)
```