Translating Chalice into SIL

Bachelor’s Thesis

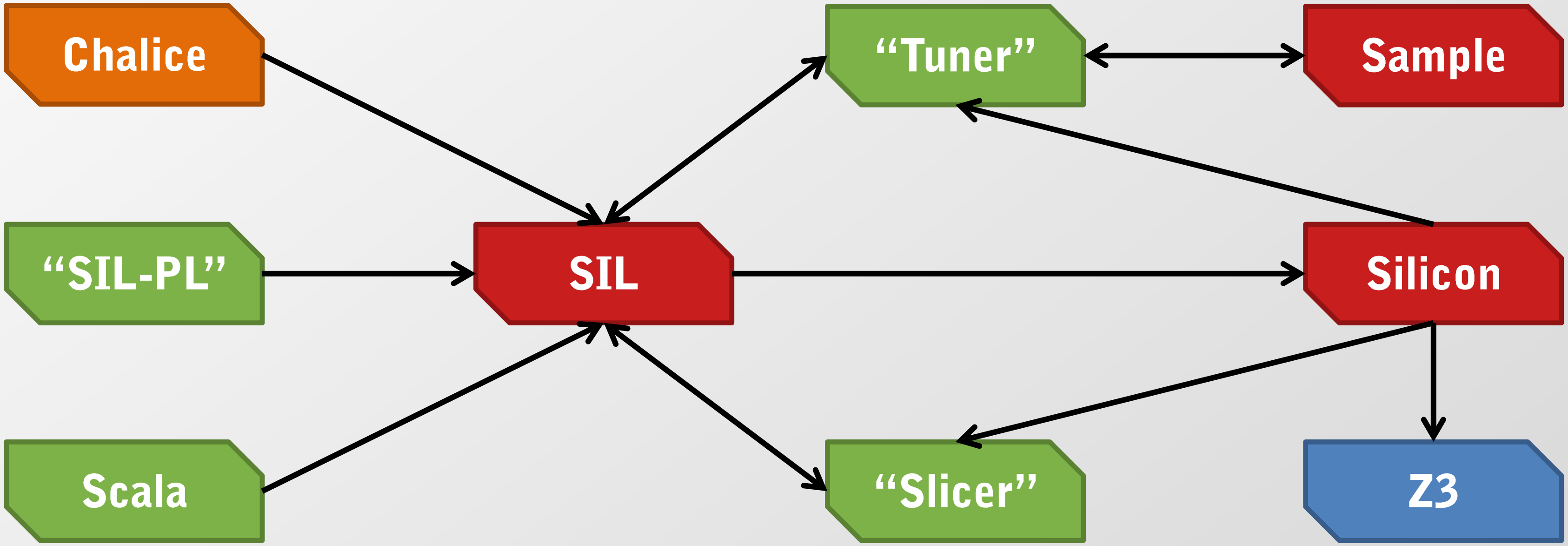
Christian Klauser

Supervisor: Dr. Alex Summers

The Semper Project

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

Semper Architecture Design



Semper Intermediate Language (SIL)

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

```
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
- Fields
- Functions
- Predicates

```
method C::cmpexc(
    this : ref,v : int,c : int)
  : (o : int)
  requires this ≠ null
  requires acc(this.C::f,write)
  ensures acc(this.C::f,write)
  ensures
    old(this.C::f == c) ⇒
      this.C::f == v
  ensures o = old(this.C::f)
```

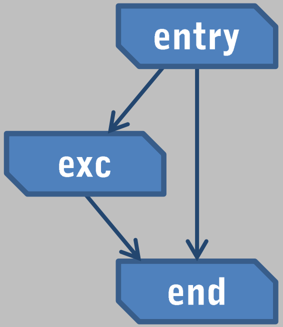
SIL Program Structure

- 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:{
  }
}
```



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

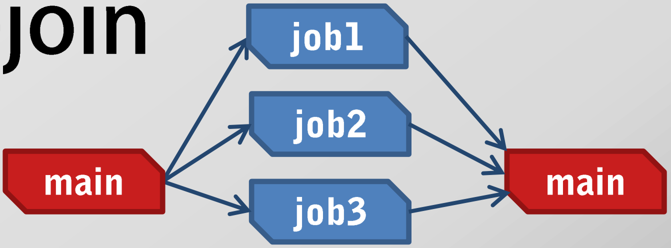
```
field C::f : int;
field L::value : int;
field L::next : ref;

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)
```

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

Transfer from caller to callee
Modular verification

Fractional Read Permissions in Chalice

Explain what fractional read permissions do
and that there is no direct SIL-equivalent

Implementing Fractional Permissions

Explain the basic idea behind the implementation.
→ select new fraction for every call-site
→ collect constraints from current amount of permissions held

Implementing Fractional Permissions #2

Show naïve translation without using map
Explain how rd(f) && rd(f) makes translation more complicated

Implementing Fractional Permissions #3

Explain translation using Map[(ref,int),Permission]

Implement Fork-Join

Another Chalice feature not present in SIL

Show how token object is assembled

→ Trick: how shadow field permissions are always linked to token.joinable

→ Current limitation: join mostly useless when not in same method as fork

QUICK DEMO

Conclusion

?

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)