Translating Chalice into SIL

Report

Christian Klauser
`klauserc@student.ethz.ch`

October 22, 2012

- Use either `acc(x.f,write)` (SIL toString) or `acc(x.f,full)` (SIL API), not both!
- Don't use abbreviated read fraction syntax `rd(x.f)`, ever.

1. Background

1.1 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 syntactical 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.

1.1.1 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 no 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. That is 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 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.

1.1.2 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⟩
```

1.1.3 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⟩
```

1.1.4 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 (only for data types in domain templates). 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 types 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*⟩ , ··· } ’]’

1.1.5 Full Expression and Term Syntax

⟨*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⟩
⟨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⟩ , ··· } )
```

References