ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Translating Scala to SIL

Master's Thesis

Bernhard F. Brodowsky

June 17, 2013

Advisors: Prof. Dr. P. Müller, M. Schwerhoff
Department of Computer Science, ETH Zürich

Abstract

In this thesis, we describe a tool which is implemented as a Scala compiler plugin that translates a Scala program to a SIL program which is then passed to a verifier. We reused experience gained with Chalice [10] and reapplied it to Scala [12] to support features like classes, methods, pure functions, the whole Implicit Dynamic Frames [17] support and the type system.

We also solved several problems that were not present in Chalice including the translation of non-pure Scala expressions to pure SIL expressions. Additionally, we found a new solution for the translation of Scala lazy vals which allows the verification of Scala programs which include lazy vals with some additional restrictions.

Contents

Chapter 1

Introduction

There exist many cases in software development, where it is acceptable to use a lot of additional resources in order to ensure the highest possible quality of the code. Sometimes, even human lives can depend on the correctness of software. Aircraft control systems and medical applications are notable examples. In those cases, it is desirable to have a guarantee for the correctness of the software by means of a mathematical proof. Doing those proofs by hand is cumbersome and error prone. Humans are almost as likely to make mistakes while creating mathematical proofs as they are while writing program code. Additionally, the proof might be correct, but the code might do something slightly different because of a typo and contain a bug nonetheless. This is why automated program verification has gained a lot of interest of the research community. Existing tools for automatic program verification include Chalice [10], Dafny [9], VeriFast [7] and Spec# [2].

One important difficulty of the verification is to make it modular, that is allowing the independent verification of different methods and components in order to keep the complexity as low as possible. There are several approaches to this problem, one of them is Implicit Dynamic Frames [17]. This will be the approach we used for our project.

Scala [12] is a modern blend of object oriented and functional programming with a powerful type system and support for the actor concurrency model. Since Scala is a very promising language with a rapidly-growing community, we chose Scala to be the language we want to verify. Our long-term goal is to provide a tool chain for automatic verification of Scala code. Note that there is already Leon, a tool for verifying pure Scala programs which is described in [18].

Our approach is to translate Scala to SIL, an intermediate verification language in the spirit of Boogie [8], but with native support for the Implicit Dynamic Frame methodology. We will describe SIL in more detail in chap-

ter 2 on page 9. The SIL code will then be passed on to a verifier. Projects which follow a similar approach exist for Chalice for which a translator to SIL and Boogie exists.

In order to allow the specification of contracts, invariants and assertions etc. to Scala code, our project includes a contracts library. There exists a project [11] which does runtime contracts checking of Scala code. The contracts library of our project is an extension of the contracts library of that project.

Our tool will be verifier independent and should be usable in combination with any SIL verifier, but the verifier we used for the development and for which it has been tested most is Silicon which basically id Syxc [16] for SIL instead of Chalice.

1.1 Motivation

This should not only be seen as a single project, but more as the start of a series of possible projects which put the efforts of the group into practice, the ultimate goal being the development of a competitive verification tool chain for Scala and possibly other programming languages which can be used in industry. It is thus crucial that this project establishes an extensible basis for future projects.

The main goal is to reuse the experience the group has gained in various other projects, the most prominent being Chalice, over the years and apply it to a real world programming language.

1.2 Main Challenges

The Scala language supports much more powerful language constructs than the relatively simple SIL language, which makes the translation more complicated than a mere dictionary-like set of replacements. In some cases, a Scala feature which doesn't exist in SIL can just be replaced by a slightly longer piece of SIL code which is equivalent to the original Scala code. An example of such a case is the translation of Scala pattern matching, which is described in more detail in section 4.10 on page 59.

In some cases however, there is a conceptual difference which makes it more complicated. For example, SIL expressions have to be pure. They are only allowed to contain binary expressions, unary expressions, pure function calls and conditional expressions. Scala expressions however can contain arbitrary side effects like method calls, assignments and loops. If a Scala expressions that contains side effects appears in a condition of a while loop in Scala, we cannot translate this expression into a SIL expression, but we

have to find a solution to take care of the side effects. Our solution to this particular problem is described in section 4.9 on page 49.

Another difficulty is the translation of Scala lazy vals to SIL. Scala lazy vals are evaluated and assigned to a field when they are first used. In SIL, we cannot always do this, for example, it is impossible to put assignments into contracts. So we have to find a way to translate lazy vals such that they can also be used in pure contexts. Additionally, the calculation of the lazy val may contain method calls or function applications with non-trivial preconditions, so we also have to make sure that those preconditions are fulfilled when the lazy val is used the first time. Our solution to those problems is described in section 4.9 on page 49.

1.3 Example

In this section we will show an example that our tool can translate and that was then verified. We show a program that calculates the sum of all integers from 0 to n for some $n \geq 0$. It calculates the sum using a loop with the loop variable i which runs from 0 to n and we added the postcondition that the result is equal to $n * (n + 1)/2$. To make the verification of the loop body possible we also added the invariant that the intermediate result is equal to $i * (i + 1)/2$ and that i stays inside the range.

```
import semper.contracts.Contracts._

class Main {

  def sum(n: Int) = {
    requires(n >= 0)
    ensures(result[Int] == n * (n + 1) / 2)
    var i = 0
    var s = 0
    while (i < n) {
      invariant(s == i * (i + 1) / 2)
      invariant(i <= n)
      i += 1
      s += i
    }
    s
  }

}
```

Figure 1.1: A simple example Scala program that uses the contracts API.

1.4 Simplified Scala

The following grammar shows the syntax of simplified Scala in EBNF. Note that white-space is mostly ignored, and that the binding strength of expressions is not modeled in the grammar for simplicity. The special repetition operator ,* is used to denote a comma-separated list.

```
scala-program ::= ( class | trait )*

        trait ::= "trait" component-name "{"
                      member*
                  "}"

        class ::= "class" component-name "{"
                      member*
                  "}"
component-name ::= ident ("[" ident ,* "]")?

         type ::= ident ("[" type ,* "]")?

  extend-list ::= "extends" type ("with" type)

       member ::= field | lazyval | function | method | predicate

        field ::= ("var" | "val") ident ":" type "=" exp

      lazyval ::= "lazy" "val" ident ":" type "=" exp

     function ::= "@pure" ident "(" formal-arg ,* ")" ":" type "=" "{"
                      precondition*
                      postcondition*
                      exp
                  "}"

    predicate ::= "@predicate" "def" ident ":" "Boolean" "=" exp

       method ::= "def" ident "(" formal-arg ,* ")" ":" type "=" "{"
                      precondition*
                      postcondition*
                      stmt*
                      exp
                  "}"

   formal-arg ::= ident ":" type

 precondition ::= "requires(" exp ")"
postcondition ::= "ensures(" exp ")"
    invariant ::= "invariant(" exp ")"

     creation ::= "new" ident ("[" type ,* "]")? "(" exp ,* ")"

  method-call ::= exp "." ident ("[" type ,* "]")? "(" exp ,* ")"

         stmt ::= "assert(" exp ")"
                | "assume(" exp ")"
                | "commit(" exp ")"
                | "fold(" loc-access ")"
                | "unfold(" loc-access ")"
                | loc-access "=" exp
                | ("var" | "val")? ident "=" exp
                | ("var" | "val")? ident "=" creation
```

```
                  // Only calls to non-pure methods
                  // are handled by this case.
                | ("var" | "val")? ident "=" method-call
                | "while" "(" exp ")" "{"
                      invariant*
                      stmt*
                  "}"
                | "if" "(" exp ")" stmt "else" stmt
                | "{" stmt* "}"

          exp ::= "this"
                | "if" "(" exp ")" exp "else" exp
                | exp "==>" exp
                | exp ("||" | "&&") exp
                | "!" exp
                | exp ("==" | "!=") exp
                | exp ("<" | "<=" | ">" | ">=") exp
                | exp ("+" | "-") exp
                | exp ("*" | "%" | "/") exp
                | ("+" | "-") exp
                | loc-access
                | integer
                | "()"
                | "null"
                | "true" | "false"
                | ident
                | "old(" a ")"
                | "(" exp ")"
                | "unfolding(" exp ")(" exp ")"

                  // Only method which represent
                  // pure functions are allowed
                | method-call

                  // The following expressions are only allowed
                  // in assertions
                | "result" "[" type "]"
                | "read(" loc-access ")"
                | "write(" loc-access ")"
                | "acc" "(" loc-access "," exp ")"
                | "uncommitted(" exp ")"

                  // The following expressions are only allowed
                  // in permission expressions
                | float
                | "rd"
                | "rd_?"
                | "rd_!"

     loc-access ::= exp "." ident

          float ::= integer "." integer

                  // An integer (specified as a regexp)
        integer ::= "[0-9]+"

                  // An identifier (specified as a regexp)
          ident ::= "[a-zA-Z$_][a-zA-Z0-9$_']*"
```

Note that the language generated by the grammar is much simpler than the actual Scala language. We left out a lot of features. Some of them, e.g. higher

order functions, because we don't support them. Others are left out because they disappear in the AST and hence we don't have to be handled. Examples include unqualified calls (i.e. without the `this` receiver), missing types and infix method calls. Several features are also left out because of simplicity, e.g. inheritance. Note that we do take care of subtyping in section 4.3 on page 25 even though we didn't include it in this simplified grammar. We also left away various border cases like methods whose body only consists of one expression without contracts even though our implementation supports those cases.

Note that the grammar allows programs that are rejected by the Scala type-checker and there are also some Scala programs which are accepted by the Scala type checker which are not valid in our context. For example, the stated grammar allows us to use predicates or permission expressions in every expression. In our project, they are only allowed inside access predicates.

Our contracts library introduces several special constructs like `fold`, `unfold`, `requires` etc. A complete list can be found in the appendix in chapter A on page 105. For the Scala compiler, those are normal method calls. It is an important part of our implementation to recognize the method calls with a special meaning and treat them specially. However, we will not go into the details here and treat them as special constructs in our grammar nonetheless to make their treatment easier to specify.

Also, in Scala, there is no difference between statements and expressions. In our grammar, we treated them separately, but in section 4.9 on page 49, we will lift that restriction and treat statements and expressions as a single concept. This also leads to expressions which might have side effects. For now, we assume that statements and expressions are not mixed and that all expressions are pure.

1.4.1 Translation Notation

During the rest of this thesis, we will assume that it is ok to use the same identifiers in Scala and SIL. This is not true in reality since Scala allows several additional symbols in identifiers which have to be removed or replaced. Sometimes, we also have to rename identifiers in order to make them unique in the SIL program. We can do this by using the identifier transformations mentioned in section 5.3 on page 100 to transform them into unique valid SIL identifiers and storing the mappings from Scala identifiers to SIL identifiers. However, we will ignore those issues in our examples to make them as simple as possible, but our translation does of course address those issues.

In order to specify our translation algorithms, we will use this grammar and the helper functions. Let $\mathcal{S}$ be the set of all possible strings and let $\mathcal{A}$ be

the set of valid Scala ASTs parsed according to our grammar including the additional restrictions and $\mathcal{B}$ be the set of valid SIL ASTs according to the grammar which will be described in chapter 2 on page 9. Let

$$\mathcal{C} = \{0, 1\} \times \mathcal{S} \times (\mathcal{A} \to \mathcal{B})$$

be the set of all possible contexts for our translation. $\mathcal{A} \to \mathcal{B}$ is the set of all possible functions from $\mathcal{A}$ to $\mathcal{B}$, so a context is a triple consisting of a boolean value which indicates if we are inside a pure method, a string containing the name of the class we are in and a function from Scala nodes to SIL nodes which is used. We will see in later sections how the components of the context are used. We introduce the function

$$\text{translate}\colon \mathcal{A} \times \mathcal{C} \to \mathcal{B}$$

for the translation.

In order to define the translation rules, we will use the translate function to define how a certain Scala tree is translated to a SIL tree. We will use expressions as in our grammars for Scala and SIL and use equations like

$$\text{translate}(\texttt{a}\ \texttt{"+"}\ \texttt{b}, c) = \text{translate}(\texttt{a}, c)\ \texttt{"+"}\ \text{translate}(\texttt{b}, c)$$

to describe the translation of a certain feature. Note that this equation alone is not sufficient since it is not clear what `a` and `b` are. Usually, we have to quantify over all possible values for which the rule applies. In this case, we could state that the rule applies for all Scala integer expressions `a` and `b` that are valid according to the grammar and all additional rules. In order to make it more concise, we will leave away the validity constraint and the requirement that they are Scala expressions, so we would just state that the rule holds for all expressions `a` and `b`.

The arguments of the translation function is a Scala AST and the right hand side is a SIL AST which may contain recursive calls. In particular, for two statements `a` and `b`. We can use `a b` as the two statements appearing next to each other like in the grammar and we can also define

$$\text{translate}(\texttt{a}\ \texttt{b}, c) = \text{translate}(\texttt{a}, c)\,\text{translate}(\texttt{b}, c).$$

The right hand side means that we recursively translate `a` and `b` to SIL nodes and form one SIL node consisting of the two translated subnodes.

We will also leave out the context argument if it is not relevant because in the majority of the cases, we will just forward the same context to all the recursive calls. We will hence only specify the context explicitly if it is used or if the recursive calls get a different context.

In order to avoid having to write a lot of very similar equations for the translation of repetitions like by `invariant`* that appear in our grammar, we only specify the translation of a single element of that list and we define that

$$\text{translate}(\texttt{""}) = \texttt{""}$$

and that for any expression `e` and any list of invariants `invs` we inductively define

$$\text{translate}(\texttt{"invariant("e ")"invs}) = \text{translate}(\texttt{"invariant"e})\,\text{translate}(\texttt{invs}).$$

If we have a non-empty list of invariants, we can apply the second rule. We can then again apply it to the returned term translate(`invs`) until that term gets empty and then we can apply the first rule to the empty string. This way, we can translate any list of invariants.

We also assume that similar translation rules exist for all other repetitions including comma separated lists. So we will only have to specify how the elements of the list are translated, but we will not specify how the lists are translated.

We also implicitly add side conditions to every translation rule we specify which state that the translated Scala node is interpreted correctly. For example, `this` is also a valid identifier, so the expression `this` could be interpreted as a local variable access. This could destroy the well-definedness of our function if we have two contradicting rules for the same Scala node. So we have to add the condition that our local variable doesn't have `this` as its name as a side condition to the translation rule for local variables. However, this would result in a lot of additional side conditions which don't give any additional insight, so we don't state them explicitly.

The top level call to initiate the translation of a Scala program `p` is

$$\text{translate}(\texttt{p}, (0, \texttt{""}, \text{id}))$$

where id is a function which is the identity function for identifiers and which maps all other Scala ASTs to some arbitrary value, for example to `""`. For the first and the second component of the context, we could choose an arbitrary value, so we just chose 0 and `""`, for the third component of the context, it is important that it maps all identifiers to themselves.

Chapter 2

The Simple Intermediate Language

The Simple Intermediate Language which we will call SIL from now on is an intermediate language that can be passed to verifiers or other tools. SIL is inspired by Boogie, which is described in [8] and also supports methods, functions and axioms. However, the type system of SIL is simpler and, for example, it doesn't support finite types. On the other hand, SIL has built-in support for a heap and objects with fields while those have to be encoded explicitly in Boogie. SIL has also been influenced by Chalice [10], namely, it incorporates the Implicit Dynamic Frames model. In particular, it supports a permission model [3] similar to Chalice's.

It also includes the possibility of adding domains for custom theories to the program. Domains can include a list of domain function definitions without body and a list of axioms which use those functions. Each domain has a domain type with an arbitrary number of type parameters associated with them which can be used in its domain functions and axioms. One could for example use a domain with one type parameter to specify a custom data structure with elements which have the type of the type parameter.

2.1 Syntax of SIL

The following grammar shows the syntax of SIL in EBNF. Note that white-space is mostly ignored, and that the binding strength of expressions is not modeled in the grammar for simplicity. Also, the grammar allows for more programs than what the SIL type-checker will accept. Again, the special repetition operator $,^*$ is used to denote a comma-separated list.

```
sil-program ::= ( domain | field | function | predicate | method )*

      domain ::= "domain" domain-name "{"
                     domain-function*
                     axiom*
                 "}"
```

```
     domain-name ::= ident | ident "[" ident ,* "]"

  domain-function ::= "unique"? "function" ident "(" formal-arg ,* ")" ":" type

       formal-arg ::= ident ":" type

            axiom ::= "axiom" ident "{"
                          exp ";"?
                      "}"

            field ::= "var" ident ":" type

         function ::= "function" ident "(" formal-arg ,* ")" ":" type
                          precondition*
                          postcondition*
                      "{" exp "}"

    precondition ::= "requires" exp ";"?
   postcondition ::= "ensures" exp ";"?
       invariant ::= "invariant" exp ";"?

        predicate ::= "predicate" ident "{" exp "}"

           method ::= "method" ident "(" formal-arg ,* ")"
                      ("returns" "(" formal-arg ,* ")")?
                          precondition*
                          postcondition*
                      "{" local-decl* stmt "}"

       local-decl ::= "var" ident ":" type

             stmt ::= (stmt ";"?)*
                    | "assert" exp
                    | "inhale" exp
                    | "exhale" exp
                    | "fold" loc-access
                    | "unfold" loc-access
                    | loc-access ":=" exp
                    | ident ":=" exp
                    | "if" "(" exp ")" "{"
                          stmt
                      "}"
                      ("elsif" "(" exp ")" "{"
                          stmt
                      "}")*
                      ("else" "{" stmt "}")?
                    | "while" "(" exp ")"
                          invariant*
                      "{" stmt "}"
                    | ident ":=" "new()"
                    | ident "(" exp ,* ")"
                    | ident ,* := ident "(" exp ,* ")"
                    | "goto" ident
                    | ident ":"
                    | "fresh" (ident*) "{"
                          stmt
                      "}"

               exp ::= exp "?" exp ":" exp
                    | exp "==>" exp
                    | exp ("||" | "&&") exp
```

```
                | "!" exp
                | exp ("==" | "!=") exp
                | exp ("<" | "<=" | ">" | ">=") exp
                | exp ("+" | "-") exp
                | exp "*" exp
                | exp ("\ " | "%") exp
                | ("+" | "-") exp
                | ident "(" exp ,* ")"
                | loc-access
                | integer
                | "null"
                | "true" | "false"
                | ident
                | "result"
                | "acc" "(" loc-access "," exp ")"
                | "forall" formal-arg ,* "::" trigger ,* exp
                | "exists" formal-arg ,* "::" exp
                | "(" exp ")"
                | "[" exp "," exp "]"
                | "perm" "(" loc-access ")"
                | "write"
                | "none"
                | "epsilon"
                | "wildcard"
                | exp "/" exp // A concrete fractional permission
                | "Seq" "[" type "]" "()"
                | "Seq" "(" exp ,* ")"
                | "[" exp ".." exp ")"
                | exp "++" exp
                | "|" exp "|"
                | exp "[" exp "]"
                | exp "[" ".." exp "]"
                | exp "[" exp ".." "]"
                | exp "[" exp ".." exp "]"
                | exp "in" exp
                | exp "[" exp ":=" exp "]"
                | "Set" "[" type "]" "()"
                | "Set" "(" exp ,* ")"
                | "Multiset" "[" type "]" "()"
                | "Multiset" "(" exp ,* ")"
                | "|" exp "|"
                | exp "union" exp
                | exp "intersection" exp
                | exp "setminus" exp
                | exp "subset" exp

    trigger ::= "{" exp ,* "}"

loc-access ::= exp "." ident

       type ::= "Int" | "Bool" | "Perm" | "Ref"
              | "Seq" "[" type "]"
              | ident
              | ident "[" type ,* "]"

      // An identifier (specified as a regular exp)
      ident ::= "[a-zA-Z$_][a-zA-Z0-9$_']*"
```

2.2 The Permission System of SIL

In SIL, there exists a permission system which requires every method or function to give a bound on the part of the heap to which it can write and a bound on the part of the heap which is read. In order to write to a field, write-permissions to that field have to be present. In order to read a field, some permissions have to be present. To express the presence of permissions to a certain field in assertions, access predicates of the form `acc(e.f, p)` have to be used. An access predicate has two arguments, the field it is referring to and an expression denoting a non-zero permission amount. The permission expression of an access predicate can be

- a positive fraction that is at most 1 with the special values `none` and `write` denoting 0 and 1 respectively,
- a wildcard that stands for an arbitrary unknown amount of permissions (arbitrary in proof obligations and unknown in assumptions), which is just guaranteed to be larger than an epsilon permission or any constant times an epsilon permission
- or an arithmetic expression of two permissions; addition, multiplication and subtraction are supported.

A method gets write permissions on every field to new objects it creates. For any other field `a.f`, we have to get the access predicate `acc(a.f, p)` for an appropriate permission `p` from somewhere. Usually, we add it to the precondition of the method, but it can also come from additional assumptions or from the postconditions of other methods we called. Often, an object is created in some method and then, the permissions for the fields are carried around with the object through contracts. If `acc(a.f, p)` appears as a proof obligation, for example in a postcondition or in the precondition of a called method, we have to prove, that we have at least a permission of `p` on `a.f` and this permission is exhaled at that point. Since calling a method takes away all the permissions that are used in the precondition, it is very common that a method mentions the same permissions in its postcondition in order to give back the permissions. A constraint on the usage of access predicate is that they can only be used in positive positions, i.e. only on the right side of implications or in conjunctions, but never negated, in disjunctions or on the left side of implications.

2.3 Abstract Predicates

An abstract predicate [15] in SIL, from now on called predicate, is a construct that takes one object as an argument. Its body is an assertion which may contain permissions. The argument of the predicate can appear in the assertion. However, the expression has to be self-framing, a concept explained in [17].

Intuitively, this means that all proof obligations of expressions that appear in the body also have to appear in the body as conjuncts. I.e. if a field appears in a predicate, an access predicate to that field and the condition that the receiver is not null have to be present. Similarly, if a function with non-trivial preconditions is used, those preconditions including all access predicates of fields that are used inside the function have to appear in a conjunct as well. The implementation of verifiers often require the proof obligations in a conjunct that appears before the usage of the expression which needs that proof obligation for well-definedness. So the code in figure 2.1 would not be valid and the code in 2.2 should be written instead.

```
predicate P(arg: Ref) {
  arg.a == arg.b
}
```

Figure 2.1: A non-self-framing SIL predicate where the permissions for the relevant fields are missing.

```
predicate P(arg: Ref) {
  arg != null &&
  acc(arg.a, wildcard) &&
  acc(arg.b, wildcard) &&
  arg.a == arg.b
}
```

Figure 2.2: A valid SIL predicate where the permissions for the relevant fields are present and appear before the usage of the field.

In the next paragraph, we will explain how the predicates are actually used by methods, but in order to be used, they have to be acquired and access predicates to them as we introduced them for fields in section 4.4 on page 31 have to be passed around through contracts. So a method which makes use of a predicate which it does not establish itself should include an access predicate on that predicate in the precondition. An example for passing around predicates in contracts can be seen in figure 2.3 on the following page.

```
method foo(arg: Ref) returns (result: Ref)
  requires acc(pred(arg), write)
  ensures acc(pred(arg), write)
{
  // Use pred in some way.
}
```

Figure 2.3: A method which uses predicate access predicates in its contracts.

In the access predicate on a predicate, it is also necessary to specify a permission for the predicate. The permissions inside the predicate are multiplied with the permission on the predicate when they are used. That is, if a predicate `pred` contains `acc(arg.a, 0.25)` and we have `acc(pred(arg), 0.5)`, the permissions we hold on `arg.a` are actually 0.125.

Three usages are possible with predicates in SIL:

Fold If a predicate is folded, the permissions that appear in the body are exhaled and the other conditions are checked. In our project, this usually happens in the constructor when the invariant is established and the permissions of all the fields have to be packaged.

Unfold In order to use the permissions and the properties of a predicate, it has to be unfolded. This means that the stored permissions are transferred to the current method and the other conditions are assumed to be true. Usually, a method that unfolds a predicate performs some actions that use those permissions or conditions, then it gets back the permissions and reestablishes the conditions and folds the predicate again.

Unfolding Unfolding is a special construct which takes an access predicate on a predicate and surrounds an expression. Note that expressions are always pure in SIL. It is in some sense equivalent to unfolding a predicate, evaluating the expression and folding the predicate again. An unfolding expression is again a pure expression. So it can also be used in pure functions, preconditions, assertions or even inside other predicates where we cannot use `fold` and `unfold` since they are statements and change the state. Note that `fold` and `unfold` don't change the program state at runtime, they only change the state from the view of the verifier since the permissions held can change.

Chapter 3

High-level Description

The goal of this project is to provide a tool chain to verify Scala code. We use an existing verifier which works on an intermediate language, SIL in our case. Our tool translates Scala to SIL and passes the SIL code to the verifier. Now the language independent task of verifying the program is separated from the language specific interpretation and translation of the source code and it enables us to add support for other languages as well.

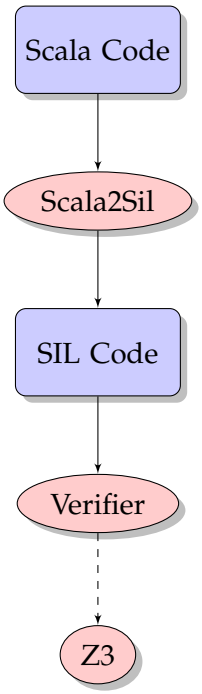

Figure 3.1: Tool chain for the verification of Scala.

Plain Scala code is hard to verify since we don't even know what it is supposed to do. Therefore, we implemented a library which contains methods such as `requires` and `invariant` which do nothing but which can be used to add specifications and informations for the verification to the Scala code. Scala code enriched with such contracts can then be passed to our tool.

3.1 Compiler Plugin

Since we don't want to write a complete parser and typechecker for the Scala language, we wanted to reuse the functionality of the Scala compiler. One way to achieve this is to write a compiler plugin which can be added to the Scala compiler. The Scala compiler then reads the Scala files containing contracts and generates an AST for each file. Those ASTs are passed to our compiler plugin one after the other.

3.2 Phases of the Translation

Our tool is implemented as a compiler plugin and works in several Scala compiler phases. The reason we need several phases is that the source Scala program might consist of several files. In such a case, the first phase is executed for all files, then the second phase is executed for all files etc. If we only had one phase, it would not be possible that a method is implemented in one file and called in another file which appears later in the compilation order. The signature is needed to translate the method call, but the implementation of the called method is encountered after the method call, so the SIL method corresponding to the called Scala method including its signature does not exist yet and the translation of the method call is not possible. So we have to collect all signatures of all files first and then we can start with the translation of the implementations for all files. We have the following phases:

Well-definedness Several well-definedness properties are checked, e.g. that permissions only appear inside contracts.

Type Collection All Scala types are collected and translated to SIL domain functions which act as type constructors.

Subtype Collection All Scala subtype relations are collected and translated to SIL domain axioms .

Signature Collection All Scala fields and method signatures are collected and the corresponding SIL fields, methods, predicates and functions are created.

Contracts Translation Predicates and contracts of methods and functions are translated.

Translation The implementations of all methods and functions are translated.

Output The resulting SIL program is assembled and written into a file or debug output, if necessary.

Verification The resulting SIL program is passed to the verifier.

Chapter 4

Main Features

4.1 Basic Translation

In this section, we will use the translation function translate to specify how the most common basic features are translated from Scala to SIL. Since we assume that the reader is familiar with most of the constructs while the others like `fold` have been described in chapter 2 on page 9, we will specify most of the rules without further explanations.

The translation of types is the first thing we handle. We translate Scala `Boolean` to SIL `Bool`, Scala `Int` to SIL `Int` and all other Scala types to `Ref`.

4.1.1 Statements

Note that we don't include rules for all the statements that are defined in our Scala grammar. Some of them require additional considerations and are treated in later chapters.

We define that for every expression `e`, we have

$$\begin{aligned}
\text{translate}(\texttt{assert(e)}) &= \texttt{"assert"}\ \text{translate}(\texttt{e}) \\
\text{translate}(\texttt{assume(e)}) &= \texttt{"inhale"}\ \text{translate}(\texttt{e}) \\
\text{translate}(\texttt{fold(e)}) &= \texttt{"fold"}\ \text{translate}(\texttt{e}) \\
\text{translate}(\texttt{unfold(e)}) &= \texttt{"unfold"}\ \text{translate}(\texttt{e}).
\end{aligned}$$

We define that for any valid left hand side of an assignment, i.e. for all local variables and for all field accesses `lhs` and for all expressions `rhs`, we have

$$\text{translate}(\texttt{lhs = rhs}) = \text{translate}(\texttt{lhs})\ \texttt{":="}\ \text{translate}(\texttt{rhs}).$$

We define that for all expressions `cond` for all lists of invariants `invs` and for all lists of statements `body`, we have

$$\begin{aligned}\text{translate}(\texttt{"while""("}\ \texttt{cond}\ \texttt{")""\{"}\ \texttt{invs}\ \texttt{body}\ \texttt{"\}"}) = \ & \texttt{"while"}\ \texttt{"("}\ \text{translate}(\texttt{cond})\ \texttt{")"} \\ & \quad \text{translate}(\texttt{invs}) \\ & \texttt{"\{"} \\ & \quad \text{translate}(\texttt{body}) \\ & \texttt{"\}"}\ .\end{aligned}$$

We define that for all expressions `cond` and for all statements `then` and `else`, we have

$$\begin{aligned}\text{translate}(\texttt{"if""("}\ \texttt{c}\ \texttt{")"}\ \texttt{then}\ \texttt{"else"}\texttt{else}) = \ & \texttt{"if("}\ \text{translate}(\texttt{c})\ \texttt{")"} \\ & \quad \text{translate}(\texttt{then}) \\ & \texttt{"else"} \\ & \quad \text{translate}(\texttt{else}).\end{aligned}$$

4.1.2 Expressions

We define that for all identifiers `ident`, for all expressions `a`, `b` and `c` and for all binary operator literals `binop` and all unary operator literal `unop` which appear in our grammar, we have

$$\begin{aligned}\text{translate}(\texttt{ident}) &= \texttt{ident} \\ \text{translate}(\texttt{a}\ \texttt{"."}\ \texttt{ident}) &= \text{translate}(\texttt{a})\ \texttt{"."}\ \texttt{ident} \\ \text{translate}(\texttt{"if""("}\ \texttt{a}\ \texttt{")"}\ \texttt{b}\ \texttt{"else"}\texttt{c}) &= \texttt{"("}\ \text{translate}(\texttt{a})\ \texttt{"?"}\ \text{translate}(\texttt{b})\ \texttt{":"}\ \text{translate}(\texttt{c})\ \texttt{")"} \\ \text{translate}(\texttt{a}\ \texttt{binop}\ \texttt{b}) &= \text{translate}(\texttt{a})\ \text{translate}(\texttt{binop})\ \text{translate}(\texttt{b}) \\ \text{translate}(\texttt{unop}\ \texttt{a}) &= \text{translate}(\texttt{unop})\ \text{translate}(\texttt{a}) \\ \text{translate}(\texttt{"old("}\ \texttt{a}\ \texttt{")"}) &= \texttt{"old("}\ \text{translate}(\texttt{a})\ \texttt{")"} \\ \text{translate}(\texttt{"unfolding("}\texttt{a}\ \texttt{")("}\texttt{b}\ \texttt{")"}) &= \texttt{"unfolding"}\ \text{translate}(\texttt{a})\ \texttt{"in"}\ \text{translate}(\texttt{b}) \\ \text{translate}(\texttt{integer}) &= \texttt{integer} \\ \text{translate}(\texttt{"this"}) &= \texttt{"this"} \\ \text{translate}(\texttt{"null"}) &= \texttt{"null"} \\ \text{translate}(\texttt{"true"}) &= \texttt{"true"} \\ \text{translate}(\texttt{"false"}) &= \texttt{"false"}\ .\end{aligned}$$

We leave away the translations of the operators since it is just a simple literal replacement.

4.1.3 Unit

A special value in the Scala language is the unit value `()`. It us commonly returned by expressions to indicate that they have no meaningful return value. We cannot just use `null` instead because the user could compare them and they should not be considered equal. Instead, we use a dummy domain with just one single domain function which returns an object which we will use as unit. The domain can be seen in figure 4.1.

```
domain UnitHelper {

  function unit(): Ref

}
```

Figure 4.1: SIl domain used to include the unit function.

After having added this domain, we can define the translation rule as

$$\text{translate}(\texttt{"()"}) = \texttt{"unit()"}\,.$$

4.2 Classes and Methods

SIL does not have the concept of classes. Methods are also quite different since they have no receiver and they don't belong to a class and hence they don't support dynamic method binding. The latter issue is left for later extensions of the project since we don't support overriding yet. The receiver is just translated as an explicit argument of the method which cannot be null. Classes are just a collection of fields and methods and we add the fields and methods of all the existing Scala classes to the SIL program.

An example is given in figure 4.2 on the next page and the translation can be seen in figure 4.3 on the following page. Note that the constructors including the field assignments are omitted in the example since they get introduced in section 4.5 on page 35. The translation of the examples is simplified because we ignore all concepts that are introduced in later sections, for example types which get introduced in section 4.3 on page 25.

```
class A {
  val a: Int = 0

  def foo(x: Int) = 2
}

class B extends A {
  var b: Int = 0

  def bar() = foo(4)
}
```

Figure 4.2: Scala Code with a two classes.

```
var A_a: Int

var B_b: Int

method A_foo(this: Ref, x: Int) returns (result: Int)
  requires this != null
{
  result := 2
}

method B_bar(this: Ref) returns (result: Int)
  requires this != null
{
  result := A_foo(this, 4)
}
```

Figure 4.3: Translation of the methods.

4.2.1 Translation Rules

The translation of classes doesn't introduce any additional SIL code, we just have to put the name of the class into the context such that all the members can use this string as a prefix. For all contexts (b, s, m) with some boolean value b, some string s and some mapping m from Scala nodes to SIL nodes, for all lists of members `mems` and for all identifiers `ident` and all comma separated identifiers lists `tparams` we define the translation rules

$$
\begin{aligned}
&\text{translate}(\texttt{"trait"}\,\texttt{ident}\ \texttt{"\{"}\texttt{mems}\ \texttt{"\}"}, (b, s, m)) \\
&\quad = \text{translate}(\texttt{mems}, (b, \texttt{ident}, m)) \\
&\text{translate}(\texttt{"trait"}\,\texttt{ident}\ \texttt{"["}\texttt{tparams}\ \texttt{"]""\{"}\,\texttt{mems}\ \texttt{"\}"}, (b, s, m)) \\
&\quad = \text{translate}(\texttt{mems}, (b, \texttt{ident}, m)) \\
&\text{translate}(\texttt{"class"}\,\texttt{ident}\ \texttt{"\{"}\texttt{mems}\ \texttt{"\}"}, (b, s, m)) \\
&\quad = \text{translate}(\texttt{mems}, (b, \texttt{ident}, m)) \\
&\text{translate}(\texttt{"class"}\,\texttt{ident}\ \texttt{"["}\texttt{tparams}\ \texttt{"]""\{"}\,\texttt{mems}\ \texttt{"\}"}, (b, s, m)) \\
&\quad = \text{translate}(\texttt{mems}, (b, \texttt{ident}, m)).
\end{aligned}
$$

Note that the type parameters are not a list of types, but a list of identifiers since this is the declaration of the type parameters, not an instantiation. They are ignored for the translation rules here and we will only use them later in section 4.13 on page 75.

For all contexts (b, s, m) with some boolean value b, some string s and some mapping m from Scala nodes to SIL nodes and for all identifiers `ident`, all expressions `exp` and all types `type` we define the translation rule for fields as

$$
\begin{aligned}
&\text{translate}(\texttt{("var"} \mid \texttt{"val")}\ \texttt{ident}\ \texttt{":"}\texttt{type}\ \texttt{"="}\texttt{exp}, (b, s, m)) \\
&\quad = \ \texttt{"var"}\ s\ \texttt{"\_"}\ \texttt{type}\ \texttt{":"}\ \text{translate}(\texttt{type}, (b, s, m))
\end{aligned}
$$

Methods have to forward the information that they are non-pure methods to their subnodes. For now, this information will be used by the translation rule for the result variable. We use the boolean flag in the context to achieve this. Note that the following translation rule is not the final version of the translation rule for methods. We will update it in section 4.3 on page 25.

For all contexts (b, s, m) with some boolean value b, some string s and some mapping m from Scala nodes to SIL nodes and for all identifiers `ident`, all formal argument lists `args`, all types `type` all precondition lists `precs`, all postcondition lists `posts`, all statement lists `stmts` and all expressions `exp`, we define the translation rule for methods that are not constructors as

```
translate( "def" ident "(" args ")" ":" type "=" "{"
    precs
    posts
    stmts
    exp
  "}" , (b, s, m))
  = "method" s "_" ident "(this: Ref," translate(args, (0, s, m)) ")"
      "returns (vResult:" translate(type, (0, s, m)) ")"
      "requires this != null"
      translate(precs, (0, s, m))
      translate(posts, (0, s, m))
    "{"
      translate(stmts, (0, s, m))
      "vResult :=" translate(exp, (0, s, m))
    "}" .
```

Note that we left out the declarations of local variables used in the body. They can be added after the translation by traversing the method body and collecting all undeclared variables.

For any identifier `arg` and any type `type`, we have

$$\text{translate}(\texttt{arg ":" type}) = \texttt{arg ":"}\ \text{translate}(\texttt{type}).$$

For the translation of the result variable, we now need the information if we are inside a pure function or a non-pure method. For all types `type`, all strings s and all mappings m from Scala nodes to SIL nodes, we define

$$\text{translate}(\texttt{"result""["}\ \texttt{type}\ \texttt{"]"}, (0, s, m)) = \texttt{"vResult"}\ .$$

We are also ready to specify the rules for method calls without type parameters For every identifier `ident`, any expression `rcv`, every method `method` and any comma separated argument list `args`, we define

```
translate(("var" | "val")?ident "="rcv "."method "("args ")")
  = ident ":=" method "(" translate(rcv) "," translate(args) ")" .
```

4.3 Basic Type System

We introduce a SIL domain `ScalaType` with the domain functions `typeOf` and `isSubType` as seen in figure 4.4. The basic axioms of the subtyping relations, i.e., reflexivity, transitivity and antisymmetry are also included. In those axioms, the first line contains the start of the quantifier including the declaration of the bound variables, the following lines contain the triggers and the last line contains the boolean expression of the axiom itself. The triggers [5] are a hint to the verifier on which expressions it should use that axiom. Without triggers, the verifier would have to try every axiom on every expression which would lead to very bad performance and possibly to matching loops. With triggers, we can give him a set of expressions which have to appear such that it makes sense to use that expression. Note that a quantified expression is also allowed to have several triggers, in our case, transitivity makes use of two triggers.

```
domain ScalaType {

  function typeOf(object: Ref): ScalaType

  function isSubType(sub: ScalaType, super: ScalaType): Bool

  axiom antisymmetry {
    forall x: ScalaType, y: ScalaType ::
    { isSubType(x, y) }
    isSubType(x, y) && isSubType(y, x) ==> (x == y)
  }

  axiom transitivity {
    forall x: ScalaType, y: ScalaType, z: ScalaType ::
    { isSubType(x, y), isSubType(y, z) }
    { isSubType(x, y), isSubType(x, z) }
    isSubType(x, y) && isSubType(y, z) ==> isSubType(x, z)
  }

  axiom reflexivity {
    forall x: ScalaType ::
    { isSubType(x, x) }
    isSubType(x, x)
  }

}
```

Figure 4.4: The ScalaType domain with the basic subtyping axioms.

For every class or trait `C` in the Scala program, we create a unique constant `unique function C(): ScalaType` which represents its type. `unique` is a special keyword for nullary functions which ensures that all functions with this keyword are mutually distinct. Additionally, we add an axiom as shown in figure 4.5 for every super class or super trait `D` of `C`.

```
axiom CsubD {
  isSubType(C(), D())
}
```

Figure 4.5: Axiom for the subtyping relation `C <: D`.

4.3.1 Adding Type Information

It doesn't suffice to have axioms for the type system, we also need to attach the type information to expressions and carry it through the program. For arguments of methods, including the receiver, and for return types of methods, we put the type information into the pre- and postcondition. We describe in section 4.4 on page 31 how we add type information for fields and in section 4.5 on page 35 how we add the type information for newly created objects. A simple Scala program is shown in 4.6 and its translation can be seen in 4.7.

```
def createHouse(p1: Person, p2: Person): House = {
  // Create house
}
```

Figure 4.6: Scala method with non-primitive arguments.

```
method createHouse(this: Ref, p1: Ref, p2: Ref)
  returns (result: Ref)
  requires this != null
  requires isSubType(typeOf(this), HouseFactory())
  requires isSubType(typeOf(p1), Person())
  requires isSubType(typeOf(p2), Person())
  ensures isSubType(typeOf(result), House())
{
  // Create house
}
```

Figure 4.7: SIL translation with contracts which contain type information.

4.3.2 Built-in Scala Types

For some special Scala types we need special axioms, for example, for the type of `null` and for the bottom type `Nothing`. Note that we don't have to add extra axioms for all built-in Scala types. The normal relations of all built-in types will be added automatically like we add them for user defined classes. But for a few classes, we need special axioms, this is why we treat the two bottom types and the three top types differently. in figure 4.8, we define the type constants for the most important built-in types. We also define some axioms for them below.

```
unique function Any(): ScalaType

unique function AnyVal(): ScalaType

unique function AnyRef(): ScalaType

unique function Unit(): ScalaType

unique function Null(): ScalaType

unique function Nothing(): ScalaType
```

Figure 4.8: Type constants for the built-in types.

```
axiom TypeOfNull {
  typeOf(null) == Null()
}

axiom TypeOfUnit {
  typeOf(unit()) == Unit()
}
```

Figure 4.9: These axiom set the types of the null literal and the unit value.

```
axiom NullSingleton {
  forall x: Ref ::
  { isSubType(typeOf(x), Null()) }
  isSubType(typeOf(x), Null()) ==> x == null
}

axiom UnitSingleton {
  forall x: Ref ::
  { isSubType(typeOf(x), Unit()) }
  isSubType(typeOf(x), Unit()) ==> x == unit()
}
```

Figure 4.10: This axiom states that no values except for the null literal and the unit value have types `Null` and `Unit`.

```
axiom AnyValSubAny {
  isSubType(AnyVal(), Any())
}

axiom AnyRefSubAny {
  isSubType(AnyRef(), Any())
}

axiom UnitSubAnyVal {
  isSubType(Unit(), AnyVal())
}
```

Figure 4.11: These axiom state the subtype relations between some built-in types.

```
axiom AnyMemberLess {
  forall x: Ref ::
  { typeOf(x) }
  typeOf(x) != Any()
}

axiom AnyValMemberLess {
  forall x: Ref ::
  { typeOf(x) }
  typeOf(x) != AnyVal()
}

axiom NothingMemberLess {
  forall x: Ref ::
  { typeOf(x) }
  typeOf(x) != Nothing()
}
```

Figure 4.12: These axiom state that there is no object whose runtime type is `Any`, `AnyVal` or `Nothing`.

```
axiom NoValRef {
  forall t: ScalaType ::
  { isSubType(t, AnyRef()) }
  { isSubType(t, AnyVal()) }
  !(t == Nothing()) ==>
    !(isSubType(t, AnyRef()) && isSubType(t, AnyVal()))
}
```

Figure 4.13: No type except for `Nothing` can be a subtype of both `AnyRef` and `AnyVal`.

```
axiom NullSubMost {
  forall t: ScalaType ::
  { isSubType(t, AnyRef()),isSubType(Null(), t) }
  isSubType(t, AnyRef()) ==>
    (t == Nothing()) || isSubType(Null(), t))
}
```

Figure 4.14: Null is a subtype of any reference type.

```
axiom NothingSubAll {
  forall t: ScalaType ::
  { isSubType(Nothing(), t) }
  isSubType(Nothing(), t)
}
```

Figure 4.15: Nothing is the bottom type and hence a subtype of every other type.

4.3.3 Translation Rules

We won't give any rules for the creation of the unique nullary domain functions and the subtyping relations since our example in figure 4.5 on page 26 is universal for non-parametrized classes. But we will update the translation rules for methods. First we define the helper function translateType which maps from unparametrized Scala types to their equivalent SIL domain function applications.

We also define the function typeCond which takes a SIL expression `e` and a Scala type `t` as arguments. If the Scala type is `Boolean` or `Int`, it returns `"true"` . For all other Scala types, we have

$$\text{typeCond}(\texttt{e},\texttt{t}) = \texttt{"isSubType(typeOf("}\ \texttt{e}\ \texttt{"), "}\ \text{translateType}(\texttt{t})\ \texttt{")"}\ .$$

We define typeConds as a function which does the same thing for a list of formal arguments, i.e. it returns the conjunction of the typeCond of all the formal arguments in the list. Now we will update the rule for methods.

For all contexts (b, s, m) with some boolean value b, some string s and some mapping m from Scala nodes to SIL nodes and for all identifiers `ident`, all formal argument lists `args`, all types `type` all precondition lists `precs`, all postcondition lists `posts`, all statement lists `stmts` and all expressions `exp`, we define the translation rule for methods that are not constructors as

```
translate( "def" ident "(" args ")" ":" type "=" "{"
    precs
    posts
    stmts
    exp
  "}" , (b, s, m))
  = "method" s "_" ident "(this: Ref," translate(args, (0, s, m)) ")"
      "returns (vResult:" translate(type, (0, s, m)) ")"
      "requires this != null"
      "requires" typeCond(this, s)
      "requires" typeConds(args)
      translate(precs, (0, s, m))
      translate(posts, (0, s, m))
      "ensures" typeCond(result, type)
    "{"
      translate(stmts, (0, s, m))
      "vResult :=" translate(exp, (0, s, m))
    "}" .
```

4.4 Permission System

In section 2.2 on page 12, we already described the SIL type system. The permission system we give the user is slightly more powerful than the permission system of SIL In our API, double constants are used for permissions and `rd_?` stands for the wildcard permission.

The main enhancement of our API are *abstract read permissions* as they exist in Chalice [6]. It is possible to give away some unknown amount `rd` of permission in order to let another method read certain values. Abstract read permissions have an important advantage over wildcard permissions: Inside a method, it is an unknown, but fixed and constant amount, so in particular, if `acc(a.f, rd)` appears in the precondition and in the postcondition of a method, it is guaranteed to be the same value. So if we have write permission on `a.f`, we can call a method that needs `acc(a.f, rd)` and gives it back afterwards and then we have restored our write permission. With wildcard permissions, this would not be possible since two occurrences of

`acc(a.f, wildcard)` are not guaranteed to stand for the same amount of permissions.

To make things easier to write down in the most common cases, we added the additional access predicates `write(a.f)` and `read(a.f)` which stand for `acc(a.f, write)` and `acc(a.f, rd)` respectively. To translate abstract read permissions, we introduce another argument `rd` to every method which holds the value of the abstract read permission inside the method. To get an appropriate value for this argument in every call, each call is surrounded by a fresh block, a special SIL construct which allows to create a permission variable `tmpRd` which can be used as the abstract read permission argument and is constrained accordingly.

Additionally, there is an unknown global permission value `rd_!` which will be explained in more detail in section 4.7 on page 46. An example in which abstract read permissions are used can be seen in figure 4.16 and its translation in figure 4.17 on the facing page.

```
def foo = {
  requires(read(f))
  ensures(read(f))
  f
}

def bar = {
  requires(write(f))
  ensures(write(f))
  foo
}
```

Figure 4.16: Scala Code with usage of abstract read permission.

```
method foo(this: Ref, rd: Perm) returns (result: Int)
  requires this != null
  requires 0 < rd && rd <= 1
  requires acc(this.f, rd)
  ensures acc(this.f, rd)
{
  result := this.f
}

method bar(this: Ref, rd: Perm) returns result: Int)
  requires this != null
  requires 0 < rd && rd <= 1
  requires acc(this.f, write)
  ensures acc(this.f, write)
{
  var tmpRd: Perm
  fresh(tmpRd) {
    result := foo(this, tmpRd)
  }
}
```

Figure 4.17: SIL code for abstract read permissions.

The unknown global constant `rd_!` is just implemented as a nullary domain function. We introduced a special domain called `PermHelper` which contains the domain function `globalRdPerm` and axioms which constrain its values. The whole domain can be seen in figure 4.18.

```
domain PermHelper {

  function globalRdPerm(): Perm

  axiom globalRdPermPositive {
    globalRdPerm() > 0
  }

  axiom globalRdPermNotFull {
    globalRdPerm() < 1
  }

}
```

Figure 4.18: SIL domain which contains the global read constant.

4.4.1 Adding Type Information

Somehow we have to carry around the type information of every field we need of every object we use. We can only use fields if we have an access predicate on it. So we can add add the type information as a conjunct after every access predicate. An example can be seen below.

```
// Inside class BackupFactory
def createBackup(): Backup = {
  requires(read(data))
  ensures(read(data))
  // Create backup
}
```

Figure 4.19: Scala method which makes use of fields.

```
method createBackup(rd: Perm, this: Ref)
  returns (result: Ref)
  requires this != null
  requires 0 < rd && rd <= 1
  requires isSubType(typeOf(this), BackupFactory())
  requires acc(this.data, rd) &&
    isSubType(this.data, Data())
  ensures isSubType(typeOf(result), Backup()) &&
    acc(this.data, rd)
  ensures isSubType(this.data, Data())
{
  // Create backup
}
```

Figure 4.20: SIL translation with type information added after every access predicate.

4.4.2 Translation Rules

We assume that we have a function double2fraction that converts Scala float representations to SIL fraction representations. We then define the translation rules for each float literal `float` as

$$\begin{aligned}
\text{translate}(\texttt{"rd_?"}) &= \texttt{"wildcard"} \\
\text{translate}(\texttt{"rd_!"}) &= \texttt{"globalRdPerm()"} \\
\text{translate}(\texttt{"rd"}) &= \texttt{"rd"} \\
\text{translate}(\texttt{float}) &= \text{double2fraction}(\texttt{float}).
\end{aligned}$$

The translation of access predicates is different depending if we are inside a non-pure method or inside a pure function. We use the boolean information we get from the translation context to distinguish between the two cases and only give the rules for non-pure methods.

We assume that we have a function fieldType which gives us the Scala type of a field. For all access locations `loc`, all permission expressions `perm` all strings s and all mappings m from Scala nodes to SIL nodes, we define

$$\begin{aligned}
&\text{translate}(\texttt{"acc" "(" loc "," perm ")"}, (0, s, m)) \\
&\quad = \texttt{"acc("}\ \text{translate}(\texttt{loc}, (0, s, m))\ \texttt{","}\ \text{translate}(\texttt{perm}, (0, s, m))\ \texttt{") \&\&"} \\
&\qquad \text{typeCond}(\texttt{loc}, \text{fieldType}(\texttt{loc})) \\
&\text{translate}(\texttt{"write" "(" loc ")"}, (0, s, m)) \\
&\quad = \texttt{"acc("}\ \text{translate}(\texttt{loc}, (0, s, m))\ \texttt{", write) \&\&"} \\
&\qquad \text{typeCond}(\texttt{loc}, \text{fieldType}(\texttt{loc})) \\
&\text{translate}(\texttt{"read" "(" loc ")"}, (0, s, m)) \\
&\quad = \texttt{"acc("}\ \text{translate}(\texttt{loc}, (0, s, m))\ \texttt{", rd) \&\&"} \\
&\qquad \text{typeCond}(\texttt{loc}, \text{fieldType}(\texttt{loc}))
\end{aligned}$$

We have to update the translation rules for methods again, but we don't state the whole rule again, we just add the additional formal argument `rd: Perm` and the precondition `requires 0 < rd && rd <= 1` to the rule. We also have to upate the translation of methods calls as

$$\begin{aligned}
&\text{translate}(\texttt{("var" | "val")?ident "="rcv "."method "("args ")"}) \\
&\quad = \texttt{"fresh(tmpRd) \{"} \\
&\qquad \texttt{ident ":="}\ \texttt{method}\ \texttt{"(rd,"}\ \text{translate}(\texttt{rcv})\ \texttt{","}\ \text{translate}(\texttt{args})\ \texttt{")"} \\
&\quad \texttt{"\}"}.
\end{aligned}$$

4.5 Constructors

Constructors require some extra work. There is nothing conceptually difficult, but we have anticipate all the transformations that the Scala compiler

does in later phases, for example we have to assign the constructor parameters to the corresponding fields. An example can be seen in figure 4.21 and its translation in figure 4.22 on the next page. Another issue here is that we need write permissions in order to assign something to a field. So we have to add `acc(this.f, write)` for every field of the class and its superclasses to the precondition of the constructor.

```
class A(val a: Int) {
  val x = 0
  def this() {
    this(0)
  }
}

class B(val b: Int) extends A(2)
```

Figure 4.21: Scala classes with a few simple constructors.

```
var A_a: Int

var A_x: Int

var B_b: Int

// Primary constructor
method A_init(this: Ref, a: Int)
  requires this != null
  requires acc(this.A_a, write)
  requires acc(this.A_x, write)
{
  this.A_a := a
  this.A_x := 0
}

// Constructor defined by def this()
method A_init_1(this: Ref)
  requires this != null
  requires acc(this.A_a, write)
  requires acc(this.A_x, write)
{
  A_init(this, 0)
}

method B_init(this: Ref, b: Int)
  requires this != null
  requires acc(this.B_b, write)
  requires acc(this.A_a, write)
  requires acc(this.A_x)
{
  this.B_b := b
  A_init(2)
}
```

Figure 4.22: SIL translation of constructors.

Typically, the access predicates on all the fields should be given to the caller, i.e. the method which creates the object and calls the constructor. This can be done by adding postconditions to the class body and those will be used as the postconditions of the primary constructor.

```
class A {
  ensures(write(a))
  var a = 0
}
```

Figure 4.23: Scala constructor with postcondition.

```
var A_a: Int

method A_init(this: Ref, a: Int)
  requires this != null
  requires acc(this.A_a, write)
  ensures acc(this.A_a, write)
{
  this.A_a := 0
}
```

Figure 4.24: SIL translation of a constructor with a postcondition.

Remember that in SIL, there exists a special statement `a := new()` to create a new object. Note that `new()` cannot be used inside arbitrary expressions, it has to be assigned directly to a local variable. In particular, it cannot be used inside pure contexts like contracts. A creation of a new object in Scala is translated into such a new statement with a following constructor call as shown in translation 4.1.

Scala	SIL

`// We have b: B`
`a = new A(2, b)`
	`a := new()`
	`inhale typeOf(a) == A()`
	`inhale isSubType(typeOf(a.b), B())`
	`A_init(a, 2, b)`

Translation 4.1: Translation of an object creation.

In the solution we chose, the creation of a new object has to be translated into a `new()` statement, inhale statements for the field types and a constructor call with no return value. Instead, it would also be possible to let the constructor contain the `new()` statement and make the newly created object the return value. Then we would translate the creation of a new Scala object into one single constructor call. However, this solution seems less desirable since calling the super constructors becomes more complicated since we only want to create one object, not one for each super call.

4.5.1 Translation Rules

We assume that there is a function

$$\text{fieldAccs}\colon \mathcal{S} \to \mathcal{B}$$

where $\mathcal{S}$ is again the set of all possible strings and $\mathcal{B}$ is again the set of valid SIL nodes. We assume that we can use this function to get the write access predicates and type constraints for all the fields of a given class. The fields of the classes are collected in advance in our implementation and hence they are available. We will omit the calculation of the function here and just assume that the information is present for all classes. Remember that we assumed that class names are unique, so this is well-defined.

Similarly, fieldAssigns gives us a list of field assignments which assign the constructor parameters to their fields for a given class. Finally inhales takes an identifier and a constructor name as an argument and gives us an inhale statement that makes sure that the given identifier gets the type of the class belonging to the given constructor and the type inhales for the fields of the class belonging to that constructor.

In the Scala AST, constructors look just like other methods, but we assume that we know for each method in the AST, if it is a constructor, a primary constructor or a normal method. We now define the translation rules for the constructors.

For all contexts (b, s, m) with some boolean value b, some string s and some mapping m from Scala nodes to SIL nodes and for all identifiers `ident`, all formal argument lists `args`, all types `type` all precondition lists `precs`, all postcondition lists `posts`, all statement lists `stmts` and all expressions `exp`, we define the translation rule for methods which are constructors, but not primary constructors as

```
translate("def" ident "("args ")"":" type "="{" precs posts stmts exp "}", (b,s,m))
  = "method" s "_" ident "(rd: Perm, this: Ref," translate(args, (0,s,m)) ")"
      "requires 0 < rd && rd <= 1"
      "requires this != null"
      fieldAccs(s)
      translate(precs, (0,s,m))
      translate(posts, (0,s,m))
    "{"
      translate(stmts, (0,s,m))
    "}"
```

while primary constructors have to assign the parameters to the fields which leads to

```
translate("def" ident "("args ")""":" type "=""{" precs posts stmts exp "}", (b,s,m))
  = "method" s "_" ident "(rd: Perm, this: Ref," translate(args, (0,s,m)) ")"
      "requires 0 < rd && rd <= 1"
      "requires this != null"
      fieldAccs(s)
      translate(precs, (0,s,m))
      translate(posts, (0,s,m))
    "{"
      fieldAssigns(s)
      translate(stmts, (0,s,m))
    "}" .
```

Note that we didn't use the return value of constructors since it is always just a unit value in the Scala AST.

For any identifier `arg` and any type `type`, we have

$$\text{translate}(\texttt{arg ":" type}) = \texttt{arg ":"}\ \text{translate}(\texttt{type}).$$

For the translation of the result variable, we now need the information if we are inside a pure function or a non-pure method. For all types `type`, all strings s and all mappings m from Scala nodes to SIL nodes, we define

$$\text{translate}(\texttt{"result""["}\ \texttt{type}\ \texttt{"]"}, (0,s,m)) = \texttt{"vResult"}\ .$$

We are also ready to specify the rules for method calls without type parameters as well as the creation of new objects without type parameters. For every identifier `ident`, any expression `rcv`, every constructor `constructor` and any comma separated argument list `args`, we define

```
translate(("var" | "val")?ident "=""new" constructor "("args ")")
  = ident ":= new()"
    inhales(ident, constructor)
    "fresh(tmpRd) {"
      constructor "(" ident "," translate(args) ")"
    "}" .
```

4.6 Pure Functions

In order to allow recursive calculations in contracts, we have to introduce pure functions as in SIL. Also, it is consistent with the principle of information hiding not to leak every detail of the implementation into the contracts which are part of the interface. Functions can be used to hide the details of certain conditions. The caller doesn't always need to know the whole precondition, it might just know that some function needs to return true. It might know that this is the case because of a previous call to another method. So it knows that the precondition is satisfied even though he doesn't know the details of the precondition.

In Scala, we don't have a special construct for pure functions, there exist only general methods, so we introduce the annotation `@pure` to declare that a certain method is pure. The translator will check that the method body is pure and translate it into a SIL function. Not every Scala method which is side-effect free can be translated into a SIL function because functions in SIL are much more restrictive, so not every side-effect free method can be annotated as `@pure`. They have to consist of a single expression that consists of integer or boolean arithmetic, conditionals, field accesses and pure function applications only. Not even the creation of new objects is allowed. An example of a pure function in Scala and its SIL counterpart is given below.

```
@pure
def fib(n: Int) = {
  requires(n >= 0)
  if (n == 0)
    0
  else if (n == 1)
    1
  else
    fib(n - 1) + fib(n - 2)
}
```

Figure 4.25: Scala code with a pure function.

```
function fib(this: Ref, n: Int): Int
  requires this != null
  requires n >= 0
{
  n == 0 ?
    0 :
    (n == 1 ?
      1 :
      fib(n - 1) + fib(n - 2))
}
```

Figure 4.26: SIL function corresponding to figure 4.25 on the preceding page.

We added a small piece of syntactic sugar to make the usage of pure functions in Scala slightly more comfortable. It is not only allowed to use a single expression, it is also possible to define vals which are used several times in the expression to make it shorter and more readable. An example can be seen in figure 4.27. The translated code can be seen in figure 4.28.

```
@pure
def foo(a: Int) = {
  val b = a * a
  b * b
}
```

Figure 4.27: A val definition in a pure function.

```
function foo(this: Ref, a: Int): Int
  requires this != null
{
  a * a * a * a
}
```

Figure 4.28: SIL function with inlined val definitions.

Note that those val definitions can not only be used before the expression, they can also be used in arbitrary blocks inside the expression.

4.6.1 Permissions in Pure Functions

Pure functions should be callable in a pure context, e.g. inside contracts. Hence we cannot surround them with a `fresh` block. So abstract read per-

missions are not an option here. But we still want to be able to call other functions which require read permissions on some field, no matter how few permissions we have on that field. The solution we chose is to just translate every permission expression to wildcard permissions in the context of pure functions. An example of a function which uses permissions can be seen in figure 4.29 and its translation in figure 4.30.

```
@pure
def foo = {
  requires(read(a))
  requires(acc(b, rd_?))
  requires(acc(c, rd_!))
  requires(acc(d, rd))
  a + b + c + d
}
```

Figure 4.29: Scala pure method that uses different kinds of permissions.

```
function foo(this: Ref): Int
  requires this != null
  requires acc(this.a, wildcard)
  requires acc(this.b, wildcard)
  requires acc(this.c, wildcard)
  requires acc(this.d, wildcard)
{
  this.a + this.b + this.c + this.d
}
```

Figure 4.30: SIL function in which every permission is translated into a wildcard.

4.6.2 Translation Rules

In order to allow val definitions inside any expression, we extend the grammar with the rules seen in figure 4.31. We also add `val-block` to the `exp` alternatives.

```
  val-def ::= "val" ident ":" type "=" exp
val-block ::= "{" val-def* exp "}"
```

Figure 4.31: Additional rules for our grammar.

Remember that the translation context has three components: A boolean value, a string and a mapping from Scala expressions to SIL expressions. We now want to use the mapping in the third component of the context to map certain variables to the expressions that were assigned to them. Remember that in the top-level call to translate the whole program, the context we used has the identity as the mapping in the third component. Remember also that we defined $\mathcal{A}$ and $\mathcal{B}$ as the sets of all valid Scala nodes and all valid Scala nodes according to our grammars. We now use the notation $m[a \to b]$ to define a new mapping in which a gets mapped to b and for all other inputs, we still map to the same value as in m. We can now use this notation to define the helper function

$$\text{addVals}\colon ((\mathcal{A} \to \mathcal{B}) \times \mathcal{A}) \to (\mathcal{A} \to \mathcal{B}).$$

This function takes two arguments, a mapping from $\mathcal{A}$ to $\mathcal{B}$ and a valid Scala node and returns another mapping from $\mathcal{A}$ to $\mathcal{B}$. We don't define it for all Scala nodes in the second node, it suffices if we define it for lists of val definitions and say that for all other inputs, the return value is arbitrary since we never call it for any other value. For the empty Scala node, we define

$$\text{addVals}(m,) = m$$

and for all identifiers `ident`, all types `type`, all lists of val definitions `vals` and all expressions `exp`, we define the recursion

$$\text{addVals}(m, \texttt{"val"}\ \texttt{ident}\ \texttt{":"}\texttt{type}\ \texttt{"="}\texttt{exp}\ \texttt{vals}) = \text{addVals}(m[\texttt{ident} \to \texttt{exp}], \mathit{vals}).$$

Note that this definition allows the newly defined expression to be used in expressions defined later.

We now replace the translation rule for identifiers in pure functions by the following. in a pure context. For all contexts $(1, s, m)$ for some string s and some mapping m from Scala nodes to SIL nodes and for all identifiers `ident`, we define

$$\text{translate}(\texttt{ident}, (1, s, m)) = m(\texttt{ident}).$$

Note that this results in the same rule as before unless `ident` has been defined in a val definition as in the grammar rule from figure `code:pure:rules` since m maps every value to itself at the beginning and only the identifiers from those val definitions are replaced.

We now want to specify the translation rule for functions. In this case, we have to tell the recursive calls via the translation context, that their subnodes appear in a pure context. For all contexts (b, s, m) with some boolean value b, some string s and some mapping m from Scala nodes to SIL nodes and for all identifiers `ident`, all formal argument lists `args`, all types `type` all precondition lists `precs`, all postcondition lists `posts`, all val definition lists

vals and all expressions exp, we define the translation rule for methods that are not constructors as

```
translate( "@pure" "def" ident "(" args ")" ":" type "="
  "{"
    precs
    posts
    vals
    exp
  "}" , (b, s, m))
  = "function" s "_" ident "(this: Ref," translate(args, (1, s, m)) "):"
    translate(type, (1, s, m))
      "this != null"
      translate(precs, (1, s, m))
      translate(posts, (1, s, m))
    "{"
      translate(exp, (1, s, addVals(m, vals)))
    "}"
```

In this case, the result variable has to be translated to the special SIL result variable for functions, so this gives us the following rule: For all types type, all strings s and all mappings m from Scala nodes to SIL nodes, we define

```
translate("result""[" type "]", (1, s, m)) = "result" .
```

We can now also add the translation rules for access predicates in pure functions. For all access locations loc, all permission expressions perm all strings s and all mappings m from Scala nodes to SIL nodes, we define

```
translate("acc" "(" loc "," perm ")", (1, s, m))
  = "acc(" translate(loc, (1, s, m)) ", wildcard) &&"
      typeCond(loc, fieldType(loc))
translate("write" "(" loc ")", (1, s, m))
  = "acc(" translate(loc, (1, s, m)) ", wildcard) &&"
      typeCond(loc, fieldType(loc))
translate("read" "(" loc ")", (1, s, m))
  = "acc(" translate(loc, (1, s, m)) ", wildcard) &&"
      typeCond(loc, fieldType(loc))
```

4.6.3 Future Work

It is possible to make functions a little less restrictive and more convenient to write for the user. In particular, the restriction that a function can only consist of one expression could be loosened a little more. However, this makes the translation more difficult since we still have to translate it into a SIL function with all the restrictions. Some additional easy cases could be possible, though.

4.7 Predicates

In Scala, predicates are methods with no argument that are annotated with `@predicate`, the receiver becomes the single argument of the translated SIL predicate. Apart from that, they are mostly equivalent to SIL predicates except that we can of course use the slightly more powerful permission system inside. Also, unfolding expressions can contain arbitrary blocks or expressions that might also have side effects if they appear in a method body. Inside assertions, pure functions or other predicates, this is not allowed, of course. If an unfolding expression in a method contains side effects, they are just extracted and put before the expression as described in 4.9 on page 49 and surrounded by `fold` and `unfold`.

4.7.1 Recursive Data Structures

An example for a using predicates for recursive data structures can be seen in figure 4.32 on the next page. We show the class of a very simple mutable list. The predicate contains all the permissions of the whole list starting from the node represented by this object. It contains the write permissions on the fields of the list, namely the pointer to the next node and the value of this cell. Either there is no next node which means that this is the last element of the list or we also have `next ne null && write(next.valid)` which contains the permissions for the rest of the list. In the constructor, we require that either the next node is null or we have `write(next.valid)` because we need the predicate for the next node to fold the predicate for this node. We also guarantee that the constructor actually establishes the predicate. In the body, we just have to fold `valid`.

```
class List(val value: Int, var next: List) {
  requires(next ne null ==> write(next.valid))
  ensures(write(valid))

  fold(valid)

  @predicate
  def valid = {
    read(value) &&
    write(next) &&
    (next ne null) ==> write(next.valid)
  }
}
```

Figure 4.32: A Scala predicate that contains all the permissions needed to access the whole list.

A method that uses the predicate could be the one in figure 4.33 which appends a new node to the current node which contains a given value. Note that the postcondition is not complete, since it doesn't make any guarantees about how the list looks after the operation, but it is enough to demonstrate the use of the predicate.

```
def insert(i: Int) {
  requires(write(valid))
  ensures(write(valid))

  unfolding(valid) {
    next = new List(i, next)
  }
}
```

Figure 4.33: A method that inserts a new node containing a given number after the current node into the list.

4.7.2 Translation Rules

For all identifiers `ident` and for all boolean expressions `exp`, we define

```
translate("@predicate""def"ident ":""Boolean""=" exp)
  = "predicate(this: Ref) {"
      "this != null &&" translate(exp)
    "}" .
```

4.8 Vals

In Scala, there exist two types of fields, vars and vals. Vars are like fields in most other object oriented programming languages, they can be read and written to. To vals, a value needs to be assigned to on declaration site and afterwards, they cannot be changed any more, so they behave like final fields in Java.

Since vals cannot be written to, it intuitively seems reasonable to assume that full permissions are not more useful than wildcard permissions, so we might as well translate any permission to a val to a wildcard permission. After this decision, the verifier can never derive that all the permissions to a val that exist in the programs state added together are at least 1, which implies that we can inhale wildcard permissions for any val that is already initialized without violating soundness. Ideally, the user would never have to write read permissions for vals since they just get inhaled automatically.

The problem is that the assumption that all the permissions present in the program add up to less than 1 only holds after the vals have been initialized. Before the initialization, we need write permissions. Hence we can only inhale wildcard permissions for the vals which are guaranteed to be already initialized. An example where inhaling access predicates on a val is not sound is shown in figure 4.34.

```
class A {
  m()
  val a = 5

  def m() {
    // Use a
  }
}
```

Figure 4.34: A problem occurs if we inhale an access predicate to `a`.

We can inhale access predicates on certain vals, however. The vals of constructor parameters are initialized before any code is executed which can use the `this` literal or otherwise, the code is rejected by the Scala compiler. We achieve this by traversing the all the translated expressions and for every constructor parameter val which is used, we prepend an inhale statement with an access predicate on that val. In the case of pure functions, we add the access predicate as a free precondition instead.

All other vals are treated like vars, i.e., we need to pass around access predicates to them in order to read them.

4.9 Side Effects

One of the first problems we encountered was the fact that an expression in Scala may have side effects while expressions in SIL are always pure. For example, the condition of a conditional statement could be a non-pure method call. It could be considered bad style to use a method call inside a condition if that method really has side effects, but pure functions in SIL cannot contain any loops and hence all Scala methods that contain loops have to be translated as non-pure methods, even if they are technically side-effect free. It definitely makes sense to call a method which uses loops in the condition of a conditional statement. In SIL, method calls can have return values, but they have to be directly assigned to a local variable and they cannot be used inside expressions. So the first approach is just to take the method calls out and assign them to temporary variables. Those temporary variables can then be used at the right point inside the SIL expression that corresponds to the Scala expression which contains this method call. An example of this approach can be seen in 4.2 where a method call inside a conditional statement is translated. Note that all translations are simplified and several things are left out in order to focus on the aspects we want to show, for example the receiver has to be given as an explicit argument in SIL while we leave it out here.

Scala	SIL

`if (foo()) {`
`  bar()`
`}`
	`tmp := foo()`
	`if (tmp) {`
	`  bar()`
	`}`

Translation 4.2: First idea for the translation of side effects.

The next thing we have to deal with are expressions with several side effects. But in most cases, we can just put the side effects in the correct order before the expression and we are fine. An example can be seen in translation 4.3,

Scala	SIL

`v = foo() +`
`  bar() * baz()`
	`tmp1 := foo()`
	`tmp2 := bar()`
	`tmp3 := baz()`
	`v := tmp1 +`
	`  tmp2 * tmp3`

Translation 4.3: Handling of several side effects.

Of course method calls are not the only expressions that can contain side effects, Scala allows blocks in arbitrary positions, too. But we can handle those just the same way:

Scala	SIL

`c = foo(`
`  {a = 2; 3},`
`  {b = 4; bar(); 5})`
	`a := 2`
	`b := 4`
	`bar()`
	`c := foo(3, 5)`

Translation 4.4: Handling of other side effects.

It gets slightly more difficult if we have expressions that do not get evaluated all the time, that is, if they are only evaluated if some other expressions fulfill certain properties. Examples are boolean binary expressions and conditional expressions. For example, in `foo() && bar()`, the right side only gets evaluated if the left side is true. In those cases, we have to put the side effects of the expressions which only get evaluated in some cases into conditional statements. This way, we can make sure that the side effects only apply if the corresponding original Scala expression gets evaluated. An example for this approach can be seen in translation 4.5.

Scala	SIL

`if (foo() && bar())`
`  2`
`else`
`  baz()`
	`tmp1 := foo()`
	`if (tmp1) {`
	`  tmp2 := bar()`
	`}`
	`if (!(tmp1 && tmp2)) {`
	`  tmp3 := baz()`
	`}`
	`tmp1 && tmp2 ? 2 : tmp3`

Translation 4.5: Handling of side effects that happen conditionally.

There is one more case we have to handle, namely side effects which change other subexpressions which should be evaluated before the side effects take place. For example, the expression `a.b + a.foo()` is suspicious. In Scala, `a.foo()` only gets executed after `a.b` is read. In SIL, the side effects of `a.foo()` have to be executed before the whole expression. But if `a.foo()` changes the value of `a.b`, we get a behavior in SIL that differs from the behavior in Scala. So we have to store `a.b` in a temporary variable before we execute the side effects of `a.foo()`. In general, we store all the expressions which might be changed by a later expressions in local variables. An example of

this approach can be seen in translation 4.6.

Scala	SIL

`a = (b + {b = 2; c}) * {c = 3; 4}`	`tmp1 := b`
	`b := 2`
	`tmp2 := tmp1 + c`
	`c := 3`
	`a := tmp2 * 4`

Translation 4.6: Handling of side effects that change previous expressions.

A difficult problem is to decide whether an expression needs to be stored into a temporary variable or not. Of course, we could just store all expressions into temporary variables, but that would make the state of the SIL program unnecessarily complex and it would also be harder to read the SIL code which may be necessary for debugging purposes. On the other hand, we have to be conservative: If any of the side effects of the later expressions could potentially change the current expression, we have to store it in a temporary variable. We chose a very simple approach: If the later expressions have no side effects, we don't have to store anything. Otherwise, we store the current expression, unless it is stable, i.e. if it contains no variables at all. So an expression that only uses constants doesn't have to be stored.

4.9.1 Adding Side Effects

While we explained how side effects of expressions are extracted, propagated, transformed and assembled, we didn't describe how they are added to the SIL program yet. We propagate them until we reach a point where the translated expression is used in a statement, e.g. if it is assigned to a local variable. We then put the side effects before the assignment statement. An example can be seen in translation 4.7. The last expression of the method which is returned by the Scala method is assigned to a result variable. So the side effects of that expressions are just added before that assignment, as well.

Scala	SIL

`a = m() + n()`	`// Extracted side effects`
	`tmp1 := m()`
	`tmp2 := n()`
	
	`// The translated assignment`
	`a := m() + n()`

Translation 4.7: Adding the side effects to the program.

The only case on which we have to spend additional thoughts are conditions of while loops because we need to make sure that the side effects of that condition are executed every time the condition is evaluated, i.e. before entering the while loop and after each run of the body. An example can be seen in translation 4.8

Scala	SIL

`while (m()) {`
`  a = a + 1`
`}`
	`// Side effects of the condition`
	`tmp1 := m()`
	`while (tmp1) {`
	`  a = a + 1`
	
	`  // Side effects of the condition`
	`  tmp1 := m()`
	`}`

Translation 4.8: Adding the side effects to the while loop condition.

4.9.2 Mixing Expressions and Statements

In Scala, there is no difference between statements and expressions, i.e. every statement can be used as an expression and every expression can be used as a statement. Using a statement as an expression is done by using the Scala expression `()` and interpreting the statement as a side effect and translate this like the previous examples. So we would get something like translation 4.9. Note that even while loops can be used as expressions with the while loop as a side effect and a return value `()`.

Scala	SIL

`assert(a == 1)`	`assert a == 1 // Side effect`
	
	`unit() // expression`

Translation 4.9: Translation of statements as expressions.

If an expression is used as a statement, e.g. in the body of a while loop, we use only the side effects of that expression in most cases. A simple example can be seen in translation 4.11 on the next page.

Scala

```
while (b) {
  m() + n()
}
```

SIL

```
while (b) {
  tmp1 := m()
  tmp2 := n()
  // tmp1 + tmp2
  // thrown away
}
```

Translation 4.10: Translation of expressions as statements.

However, we cannot just throw away the expression, if it still contains proof obligations. If the user calls a function for which he doesn't fulfill the precondition or if he uses a field, for which he doesn't have any permissions, the verification should fail. So we will have to include those expressions into the SIL code. In those cases, we assign the expressions to dummy variables. An example of such a case can be seen in translation 4.11.

Scala

```
while (b) {
  m() + a.b
}
```

SIL

```
while (b) {
  tmp1 := m()
  // Expression cannot
  // be thrown away
  // and is assigned to
  // a dummy variable.
  dummy := tmp1 + a.b
}
```

Translation 4.11: Translation of expressions as statements.

4.9.3 Translation Rules

We define the additional translation function

$$\text{translateNonPure}\colon \mathcal{A} \times \mathcal{C} \times \mathbb{N} \to \mathcal{B} \times \mathcal{B}^*$$

which takes the same arguments as translate, but returns a list of SIL statements besides the translated node and takes the position of the current node in the pre-order traversal of the body of the current method as an additional argument. However, we will not perform calculations for the positions of the subnodes in our rules. Instead, we will just use the variables $n_0, n_1, \ldots, n_k$ as placeholders for the positions of subnodes. Note that if we are given the position of the parent node, it is in fact possible to calculate the positions of all the subnodes in the pre-order traversal without any additional information.

We assume that we can use the functions

$$\text{exp}\colon \mathcal{B} \times \mathcal{B}^* \to B \text{ with } (a, b) \mapsto a$$

and

$$\text{stmts}\colon \mathcal{B} \times \mathcal{B}^* \to B^* \text{ with } (a,b) \mapsto b$$

to extract the first and the second element of a pair returned by translateNonPure.

Additionally, we define the helper functions

$$\text{isUnchangable}\colon \mathcal{A} \to \{0,1\}$$

which returns 1 aka true iff the input is a constant expression or the `this` literal. We also assume that we have

$$\text{isEmpty}\colon \mathcal{B}^* \to \{0,1\}$$

which returns 1 aka true for an empty list and 0 aka false otherwise. We also define the function

$$\text{tmpVar}\colon \mathbb{N} \to \mathcal{A} \text{ with } n \mapsto \texttt{"tmp\_"}\ n$$

which returns a different temporary variable for each position.

Building upon those helper functions, we can define

$$\text{saved}\colon \mathcal{B} \times \mathcal{B}^* \times \mathbb{N} \to \mathcal{B}$$

and

$$\text{save}\colon \mathcal{B} \times \mathcal{B}^* \to \mathcal{B}$$

which help us to save expressions as we described in the examples.

Intuitively, $\text{saved}(a,b,n)$ and $\text{save}(a,b,n)$ return a and an empty string respectively, if the side effects described by b cannot affect the expression a and a temporary variable and an assignment that stores a in that temporary variable otherwise. Formally, we have

$$\text{saved}(a,b,n) = \text{if isUnchangable}(a) || \text{isEmpty}(b,n) \text{ then } a \text{ else tmpVar}(a,n)$$

and

$$\text{save}(a,b,n) = \text{if isUnchangable}(a) || \text{isEmpty}(b,n) \text{ then } \texttt{""} \text{ else tmpVar}(a,n)\ \texttt{":="}\ a.$$

In order to allow non-pure expressions and to remove the difference between statements and expressions, we have to update the grammar from section 1.4 on page 4. We replace the rule for methods as

```
method ::= "def" ident "(" formal-arg,* ")" ":" type "=" "{"
               precondition*
               postcondition*
               nonpure-exp*
           "}"
```

We won't specify the whole rules for the translation of methods and constructors again, we think that it suffices to state that for all contexts c the body, i.e. the `nonpure-exp`* part will be translated as

$$\begin{aligned}&\mathrm{exp}(\mathrm{translateNonPure}(\texttt{body}, c, 0))\\&\texttt{"vResult :="}\ \mathrm{stmts}(\mathrm{translateNonPure}(\texttt{body}, c, 0)).\end{aligned}$$

For constructors, we again leave away the result assignment. So we just add the side effects inside the constructed SIL method body and assign the final expression at the end. Note that we used 0 as the position argument, since the position describes the position of the node in pre-order traversal of the body of the current method. Since we only use it to create unique temporary variables for expressions, this suffices because it yields temporary variables which are unique inside the method body even though different methods can get the same local variables.

We now have to define the grammar for non-pure expressions as the combination of the rules for statements and for expressions. We get the following:

```
nonpure-exp ::= exp
              | "assert(" exp ")"
              | "assume(" exp ")"
              | "commit(" exp ")"
              | "fold(" loc-access ")"
              | "unfold(" loc-access ")"
              | nonpure-exp "." ident "=" nonpure-exp
              | creation
              | method-call
              | "while" "(" nonpure-exp ")" "{"
                    invariant*
                    nonpure-exp*
                "}"
              | "{" nonpure-exp* "}"
              | "if" "(" nonpure-exp ")" nonpure-exp "else" nonpure-exp
              | nonpure-exp "==>" nonpure-exp
              | nonpure-exp ("||" | "&&") nonpure-exp
              | "!" nonpure-exp
              | nonpure-exp ("==" | "!=") nonpure-exp
              | nonpure-exp ("<" | "<=" | ">" | ">=") nonpure-exp
              | nonpure-exp ("+" | "-") nonpure-exp
              | nonpure-exp ("*" | "\%" | "/") nonpure-exp
              | ("+" | "-") nonpure-exp
              | nonpure-exp "." ident
              | "(" nonpure-exp ")"
              | "unfolding(" exp ")(" nonpure-exp ")"
              | "{" nonpure-exp* "}"
```

Note that the old definition of pure `exp` still appears sometimes, it is also used as one alternative for `nonpure-exp` such that we don't have to include all common things like integer constants to the grammar rule. From now on, we will call the expressions generated by the old `exp` rule pure expressions.

Now we can redefine all the translation rules for non-pure expressions. In order to reuse the translation rules of the terminal expressions like integers

etc, we define for all pure expressions `e`

$$\text{translateNonPure}(\texttt{e}, c, n) = (\ \texttt{""}\ , \text{translate}(\texttt{e}, c)).$$

For items that have originally been translated as basic statements, it is quite straightforward and we define for all pure expressions `e`, all contexts `c` and all positions $n \in \mathbb{N}$

$$\begin{aligned}
\text{translateNonPure}(\texttt{assert(e)}, c, n) &= (\ \texttt{"assert"}\ \text{translate}(\texttt{e}, c),\ \texttt{"unit()"}\ ) \\
\text{translateNonPure}(\texttt{assume(e)}, c, n) &= (\ \texttt{"inhale"}\ \text{translate}(\texttt{e}, c),\ \texttt{"unit()"}\ ) \\
\text{translateNonPure}(\texttt{fold(e)}, c, n) &= (\ \texttt{"fold"}\ \text{translate}(\texttt{e}, c),\ \texttt{"unit()"}\ ) \\
\text{translateNonPure}(\texttt{unfold(e)}, c, n) &= (\ \texttt{"unfold"}\ \text{translate}(\texttt{e}, c),\ \texttt{"unit()"}\ ).
\end{aligned}$$

The rules for expressions are a little bit more difficult to specify. For the rest of this section, we leave away the quantifications and assume that we always quantify over all contexts z and over all non-pure expressions a, b and c and all pure expressions e.

$$\begin{aligned}
&\text{translateNonPure}(\texttt{"if"}\ \texttt{"("}\ \texttt{a}\ \texttt{")"}\ \texttt{b}\ \texttt{"else"}\ \texttt{c}, z, n) \\
&\quad = (\\
&\qquad \text{stmts}(\text{translateNonPure}(\texttt{a}, z, n_1)) \\
&\qquad \text{save}(\text{exp}(\text{translateNonPure}(\texttt{a}, z, n_1)), \\
&\qquad\quad \text{stmts}(\text{translateNonPure}(\texttt{b}, z, n_2))\ \text{stmts}(\text{translateNonPure}(\texttt{c}, z, n_3)), n_1) \\
&\qquad \texttt{"if("}\ \text{saved}(\text{exp}(\text{translateNonPure}(\texttt{a}, z, n_1)), \\
&\qquad\quad \text{stmts}(\text{translateNonPure}(\texttt{b}, z, n_2))\ \text{stmts}(\text{translateNonPure}(\texttt{c}, z, n_3)))\ \texttt{") \{"} \\
&\qquad\quad \text{stmts}(\text{translateNonPure}(\texttt{b}, z, n_2)) \\
&\qquad \texttt{"\} else \{"} \\
&\qquad\quad \text{stmts}(\text{translateNonPure}(\texttt{c}, z, n_2)) \\
&\qquad \texttt{"\}"} \\
&\qquad , \\
&\qquad \texttt{"("}\ \text{saved}(\text{exp}(\text{translateNonPure}(\texttt{a}, z, n_1)), \\
&\qquad\quad \text{stmts}(\text{translateNonPure}(\texttt{b}, z, n_2))\ \text{stmts}(\text{translateNonPure}(\texttt{c}, z, n_3)))\ \texttt{"?"} \\
&\qquad\quad \text{exp}(\text{translateNonPure}(\texttt{b}, z, n_2))\ \texttt{":"} \\
&\qquad\quad \text{exp}(\text{translateNonPure}(\texttt{c}, z, n_2))\ \texttt{")"} \\
&\qquad)
\end{aligned}$$

What happens is exactly what we described in the examples. First we put the side effects of the condition, then we save it, if it is affected by the later side effects. Then we use the potentially saved expression to execute the side

effects of the branches conditionally. At the end, we return the conditional expression consisting of the potentially saved condition and the expressions of the branches. For conjunctions we define

```
translateNonPure(a "&&" b, z, n)
  = (
    stmts(translateNonPure(a, z, n1))
    save(exp(translateNonPure(a, z, n1)),
      stmts(translateNonPure(b, z, n2)) stmts(translateNonPure(c, z, n3)), n1)
    "if(" saved(exp(translateNonPure(a, z, n1)),
      stmts(translateNonPure(b, z, n2)) stmts(translateNonPure(c, z, n3))) ") {"
      stmts(translateNonPure(b, z, n2))
    "}"
   ,
    saved(exp(translateNonPure(a, z, n1)), stmts(translateNonPure(b, z, n2)) stmts(translateNonPure(c,
      exp(translateNonPure(b, z, n2))
   )
```

We execute the side effects of the left hand side first, then we save the left hand side, if necessary, we conditionally execute the side effects of the right hand side and we return the translated conjunction as the expression. We will not specify the rules for implication and disjunction because they are almost identical. We just have to replace the operator at the end and for disjunctions, we also have to negate the guards for the side effects of the second expression. For function calls, we quantify over all non-pure expressions r, all function names f, all $k \in \mathbb{N}$ and all sequences of non-pure expressions $a_1, \ldots, a_k$.

$$
\begin{aligned}
&\text{translateNonPure}(r\ \texttt{"."}\ f\ \texttt{"("}\ a_1\ \texttt{","}\ \ldots\ \texttt{","}\ a_k\ \texttt{")"}, z, n) \\
&\quad = (\\
&\qquad \text{stmts}(\text{translateNonPure}(r, z, n_0)) \\
&\qquad \text{save}(\text{exp}(\text{translateNonPure}(r, z, n_0)), \\
&\qquad\quad \text{stmts}(\text{translateNonPure}(a_1, z, n_1)) \ldots \text{stmts}(\text{translateNonPure}(a_k, z, n_k)), n_0) \\
&\qquad \text{stmts}(\text{translateNonPure}(a_1, z, n_1)) \\
&\qquad \text{save}(\text{exp}(\text{translateNonPure}(a_1, z, n_1)), \\
&\qquad\quad \text{stmts}(\text{translateNonPure}(a_2, z, n_2)) \ldots \text{stmts}(\text{translateNonPure}(a_k, z, n_k)), n_1) \\
&\qquad \ldots \\
&\qquad \text{stmts}(\text{translateNonPure}(a_{k-1}, z, n_{k-1})) \\
&\qquad \text{save}(\text{exp}(\text{translateNonPure}(a_{k-1}, z, n_{k-1})), \\
&\qquad\quad \text{stmts}(\text{translateNonPure}(a_k, z, n_k)), n_{k-1}) \\
&\qquad \text{stmts}(\text{translateNonPure}(a_k, z, n_k)) \\
&\quad , \\
&\quad f\ \texttt{"("} \\
&\qquad \text{saved}(\text{exp}(\text{translateNonPure}(r, z, n_0)), \\
&\qquad\quad \text{stmts}(\text{translateNonPure}(a_1, z, n_1)) \ldots \text{stmts}(\text{translateNonPure}(a_k, z, n_k)), n_0)\ \texttt{","} \\
&\qquad \text{saved}(\text{exp}(\text{translateNonPure}(a_1, z, n_1)), \\
&\qquad\quad \text{stmts}(\text{translateNonPure}(a_2, z, n_2)) \ldots \text{stmts}(\text{translateNonPure}(a_k, z, n_k)), n_1)\ \texttt{","} \\
&\qquad \ldots\ \texttt{","} \\
&\qquad \text{saved}(\text{exp}(\text{translateNonPure}(a_{k-1}, z, n_{k-1})), \\
&\qquad\quad \text{stmts}(\text{translateNonPure}(a_k, z, n_k)), n_{k-1}) \\
&\qquad \text{exp}(\text{translateNonPure}(a_k, z, n_k))\ \texttt{")"} \\
&\quad)
\end{aligned}
$$

The formal rule is huge, but it just tells us that the side effects are taken out, saved depending on the later side effects and then the expression is formed with the (possibly saved) side effects. We will not write down the rule for method calls and object creations in non-pure expressions since it is very similar to the rule for function applications. The only difference is that the call is surrounded by a `fresh` block and put to the end of the side effects while a temporary variable is used as the expression.

We also omit the rules for all other expressions because they can all be derived from the rule for function applications. For example, the remaining binary expressions be handled just like function applications with one argument (plus receiver), where we replace the expression at the end by a binary expression instead of a function application.

One construct that requires additional attention is the while loop. For all lists of non-pure statements s we define

$$
\begin{aligned}
&\text{translateNonPure}(\texttt{"while"}\ \texttt{"("}\ a\ \texttt{")"}\ \texttt{"\{"}\ s\ \texttt{"\}"}, z, n)\\
&= \text{stmts}(\text{translateNonPure}(a, z, n_1))\\
&\quad \texttt{"while("}\ \text{exp}(\text{translateNonPure}(a, z, n_1))\ \texttt{") \{"}\\
&\qquad \text{stmts}(\text{translateNonPure}(s, z, n_2))\\
&\qquad \text{stmts}(\text{translateNonPure}(a, z, n_1))\\
&\quad \texttt{"\}"}
\end{aligned}
$$

Note that we left out the translation of the invariants and the potential assignment of the expression of the body. The side effects of the condition are executed once before the body and at the end of the body.

4.9.4 Future Work

There is no really important extension left since our work is complete here. However, there is still room for small improvements to make the translated SIL program slightly more readable and simpler to verify. One thing that could be improved is the algorithm which decides if an expression needs to be stored in a temporary variable or not.

4.10 Pattern Matching

Pattern matching is a very important concept in Scala. It can be used to deal with algebraic data types in a very elegant way. In SIL, there is no pattern matching yet and if it will exist in the future, so we have to translate it to nested conditional expressions.

If none of the cases applies of the pattern matching, a Match Error is thrown in Scala. In SIL, we decided that we should add a proof which states obligation that this never happens because we don't have support for exceptions yet and it probably indicates an error in the original Scala program. In methods, we can just write `assert(false)` in this last branch which should never be entered. But since we also need this inside pure functions, we introduced the artificial pure function `fail()` with false as the precondition. So if the branch can be entered, the verification will fail.

Scala

```
a match {
  case 0 => 3
  case _ if a >= 1 => 5
}
```

SIL

```
a == 0 ?
  3 :
  (a >= 1 ?
    5 :
    fail())
```

Translation 4.12: Translation of very simple pattern matching.

The Scala compiler claims that the pattern from the example is not exhaustive. However, the verifier might know more. For example, if the precondition of the function or method in which the code snipped appears has the precondition `a >= 0`, one of the first two cases will always match and hence the verifier will accept this pattern.

Of course, we also allow side effects inside the bodies of the pattern matching if it is used inside a method. Side effects are handled according to the principles described in 4.9 on page 49. If side effects appear in a condition or in the body, they are put before the expression. An example is given in translation 4.13.

Scala

```
a match {
  case 0 => { b = 17; 1 }
  case _ => 2
}
```

SIL

```
if (a == 0) {
  b := 17
}
a == 0 ? 1 : 2
```

Translation 4.13: Translation of a pattern matching with some side effects in bodies.

Even the conditions are allowed to contain side effects. Such an example can be seen in translation 4.14.

Scala

```
a match {
  case 0 if b.foo() => 1
  case _ => 2
}
```

SIL

```
if (a == 0) {
  tmp1 := foo(b)
}
a == 0 && tmp1 ? 1 : 2
```

Translation 4.14: Translation of a pattern matching with some side effects in conditions.

If the side effects of a body change an earlier condition, that condition has to be saved.

Scala

```
a match {
  case 1 if b => 4
  case _ => { b = false; 5 }
}
```

SIL

```
tmp1 := a == 1 && b
if (tmp1) {
} else {
  b := false
}
tmp1 ? 4 : 5
```

Translation 4.15: Translation of a pattern matching with side effects in an action which changes the condition.

Analogously, if the side effects of a later condition might change an earlier condition, that condition is saved. An example can be seen in translation 4.16

Scala

```
a match {
  case 1 if b.c => 4
  case _ if b.foo() => 5
  case _ => 6
}
```

SIL

```
tmp1 := a == 1 && b.c
if (tmp1) {
} else {
  tmp2 := b.foo()
}
tmp1 ?
  4 :
  (tmp2 ? 5 : 6)
```

Translation 4.16: Translation of a pattern matching with side effects in a body that changes another condition.

Up to now, we only showed very basic examples of pattern matching. But it is also very common to use the type information in Scala. The encoding of types will be described in 4.3 on page 25. An example of a Scala pattern matching that uses type information can be seen in 4.17

Scala

```
a match {
  case _: C => 5
  case _ => 6
}
```

SIL

```
(isSubType(typeOf(a), C()) &&
  a != null) ?
  5 :
  6
```

Translation 4.17: Translation of a pattern matching which contains type related matches.

A very useful way to use pattern matching in Scala is using case classes. For example, the user may define case classes as seen in figure 4.35 on the following page.

```
case class C(f: Int, g: Int)
case class D(h: Int)
```

Figure 4.35: Scala case classes

Then they can use `C(x, y)` as a pattern which matches if the matched expression has type `C` and is non-null. In the block of that pattern, `x` and `y` are bound to the value of the fields `f` and `g` respectively.

The problem here is that we need the permissions to the fields, in this example `f` and `g`. But the permissions we need depend on the type the object has at runtime. In particular, we would not even be allowed to write, for example, `read(a.f)` if we don't know that `a` has type `C`.

```
// User cannot require read(a.f) here.
a match {
  case C(x, y) => {
    // Here we need permissions on a.f and a.g.
    // Otherwise, we cannot use x and y.
  }
}
```

Figure 4.36: Scala pattern matching which contains a case class matches.

But as we explained in 4.8 on page 48, we can just inhale permissions for fields if they are constructor parameter vals, which is usually the case for case fields. So the user just gets the permissions for the case fields for free. For the rare case of case vars, we cannot do this, so in those rare cases, the user will have to deal with the problem himself.

In the case of functions, inhale statements are not possible, since the body can only consist of a single expression. But in that case, we can introduce the additional permissions via a special construct inside the precondition that doesn't have to be proved when the function is called, but which can be assumed inside the body. In SIL, there exists a special expression called Inhale-Exhale-expression which has two parts where the first one is used whenever the expression is used as an assumption and the second one is used whenever the expression is used as a proof obligation. So we put `true` into the second part and the access predicates on all the needed fields into the first part.

However, it is not yet possible to use additional assumptions inside contracts. The Inhale-Exhale-expression doesn't help us here since we always need the

permissions to the fields used inside contracts no matter if the contracts appear as a proof obligation or as an assumption.

An example of a pattern matching which makes use of case classes and its translation can be seen in figure 4.37. Note that we assumed that this expression appears in a method, which means that we can inhale the additional permissions for the case fields. The translation can be seen in figure 4.38.

```
a match {
  case C(x, y) => x + y
  case D(x) => x
  case _ => 5
}
```

Figure 4.37: Scala pattern matching which contains case class matches.

```
if (isSubType(typeOf(a), C()) && a != null) {
  inhale(acc(a.f, wildcard))
  inhale(acc(a.g, wildcard))
} elsif (isSubType(typeOf(a), D()) && a != null) {
  inhale(acc(a.h, wildcard))
}
isSubType(typeOf(a), C()) && a != null ?
  a.f + a.g :
  (isSubType(typeOf(a), D()) && a != null) ?
    a.h :
    5)
```

Figure 4.38: Translation of a pattern matching which contains case class matches.

4.10.1 Translation Rules

In order to be able to add translation rules for pattern matching, we have to extend the grammar, first.

```
pattern ::= ident
          | caseclass
          | literal
          | ident ":" type
          | "_" ":" type
          | "_"

literal ::= "true"
          | "false"
          | "null"
          | integer
```

```
    caseclass ::= ident "(" ident* ")"
 nonpure-case ::= "case" pattern "if" nonpure-exp "=>" nonpure-exp
nonpure-match ::= ident "match" "{" case* "}"
         case ::= "case" pattern "if" exp "=>" exp
        match ::= ident "match" "{" case* "}"
```

Note that our grammar is much more restrictive than the actual Scala grammar. We don't allow nested patterns and the additional condition with `"if"` `exp` is not really necessary in Scala. We can assume that the condition is just `"true"` in those cases where an additional condition is not needed.

Additionally, we include

```
"throw" "new" "MatchError" "(" ")"
exp "." "isInstanceOf" "[" type "]"
```

to the rules of `exp` and for all expressions `e`, we translate them as

$$\begin{aligned}&\text{translate}(\texttt{"throw"}\,\texttt{"new"}\,\texttt{"MatchError"}\,\texttt{"("}\,\texttt{")"})\\&\quad = \texttt{"fail()"}\\&\text{translate}(\texttt{e}\ \texttt{"."}\,\texttt{"isInstanceOf"}\,\texttt{"["}\texttt{type}\ \texttt{"]"})\\&\quad = \texttt{"isSubType(typeOf("}\ \text{translate}(\texttt{exp})\ \texttt{"),"}\ \text{translate}(\texttt{type})\ \texttt{")"}\end{aligned}$$

Note that we only allow an immutable local variable as the matched expression. This simplifies the transformations and actually, it is the only thing we have to support because the Scala compiler always transforms match expressions in such a way that they fulfill this requirement. The matched expression is always assigned to an internal local variable by the Scala compiler and instead, the local variable is used as the matched expression. Note that this doesn't make it impossible for pattern matches to appear in pure contexts. Remember that the translation of pure expressions can handle val defs.

Instead of translating pattern matching to SIL directly, we translate it to nested Scala if statements. Those will then be translated again. This allows us to avoid having to state the complicated handling of side effects again.

We define the transformation function

$$\text{transform}\colon \mathcal{A} \to \mathcal{A}.$$

for the transformation and the helper functions

$$\text{patCond}\colon \mathcal{A} \times \mathcal{A} \to \mathcal{A}$$

and

$$\text{patStmts}\colon \mathcal{A} \to \mathcal{A}$$

for extracting the condition and the additional statements of a pattern.

The function patCond takes the matched identifier and a pattern as arguments. It returns the condition that this pattern matches the matched identifier. For all identifiers `a` and `ident`, all literals `lit`, all types `type`, all comma separated identifier lists `args`, we define

$$\begin{aligned}
\text{patCond}(\texttt{a}, \texttt{lit}) &= \texttt{a}\ \texttt{"=="}\ \texttt{lit} \\
\text{patCond}(\texttt{a}, (\texttt{"_"}\ \texttt{ident})\ \texttt{":"}\texttt{type}) &= \texttt{a}\ \texttt{".isInstanceOf["}\texttt{type}\ \texttt{"]"} \\
\text{patCond}(\texttt{a}, \texttt{ident(}\ \texttt{args}\ \texttt{)}) &= \texttt{a}\ \texttt{".isInstanceOf["}\texttt{type}\ \texttt{"]"}
\end{aligned}$$

for all other arguments, patCond just returns `"true"` .

The function patStmts assigns all variables that are bound in the pattern to their value. In the case of a case class pattern, all fields are assigned, in the case of an identifier or the `a ":" type` pattern, the matched expression is assigned to the variable from the pattern. In other cases, nothing is done. We don't specify this function formally since it is clear what it does, but it would require some more work to specify formally how the case fields are assigned.

The base case of transform is defined as

$$\begin{aligned}
&\text{transform}(\texttt{a}\ \texttt{"match""\{"}\ \texttt{"\}"}) \\
&\quad = \texttt{"throw new MatchError"}\ .
\end{aligned}$$

and for the inductive case, we define for all case lists `cases`, for all identifiers `a`, for all patterns `pattern` and for all expressions `b` and `c` we define

$$\begin{aligned}
&\text{transform}(\texttt{a}\ \texttt{"match""\{"} \\
&\quad \texttt{"case"}\ \texttt{pattern}\ \texttt{"if"}\texttt{b}\ \texttt{"=>"}\texttt{c} \\
&\quad \texttt{cases} \\
&\ \texttt{"\}"}\ ) \\
&= \texttt{"if ("}\ \text{patCond}(\texttt{a}, \text{transform}(\texttt{pattern}))\ \texttt{"\&\&"}\ \text{transform}(\texttt{b})\ \texttt{") \{"} \\
&\qquad \text{patStmts}(\text{transform}(\texttt{pattern})) \\
&\qquad \text{transform}(c) \\
&\quad \texttt{"\} else \{"} \\
&\qquad \text{transform}(\texttt{exp}\ \texttt{"match"}\ \texttt{"\{"}\ \texttt{cases}\ \texttt{"\}"}\ ) \\
&\quad \texttt{"\}"}\ .
\end{aligned}$$

We define the same rule for non-pure expressions as well. For all other expressions, transform just recursively transforms subnodes and replaces them, if necessary. For the translation of a Scala program that contains pattern matching, we apply transform to the program first to translate the pattern matching to nested if expressions and then we translate the resulting program.

4.11 Lazy Vals

It is very common in programs that initialization of components takes quite some time. This can lead to long startup times for programs, because not all components might be used in every execution. Scala provides a very elegant way to address this problem. It is possible to declare an immutable field as `lazy val a = m()` instead of `val a = m()` which causes the expression on the right hand side, `m()` in our example, to be evaluated when `a` is used the first time. If `a` is never used, the right hand side is never evaluated.

One possibility would be to let the user track for every lazy val if it has been evaluated already. Then it would be easy to just evaluate it at the right point in SIL. But that would not be an acceptable solution since it destroys all the benefits of lazy vals.

4.11.1 Restrictions

Usually, the point of lazy vals is that it doesn't matter when they are evaluated. The expression on the right hand side of a lazy val should stay constant, i.e. it should always return the same result no matter when it is evaluated. Also, the calculation often doesn't contain any side effects. Usually, the user wants to have control over the side effects that get executed and when they are executed. If there are side effects in the calculation of the lazy val, it is often intransparent when they are executed. So it is a reasonable restriction to forbid side effects in the calculation of lazy vals. It may be useful in exceptional cases, but in our model, we will forbid it.

The idea is that we restrict the right hand side of lazy vals to calculations that are side-effect free and whose result is the same, no matter when we evaluate it. Then, we can evaluate the calculation as early as possible and save it into a field in the translated SIL program.

Making an Expression Constant

We found a good way to guarantee that an expression stays constant: We just exhale a part of all the permissions needed to calculate it, then it can never change. For all fields, on which the calculation depends, we exhale

wildcard permissions on that field are needed. Let us consider the expression `a.f + a.g`. The fields `a.f` and `a.g` are used in an expression. We assume that we have an access predicate on the two fields. We now execute the code from figure 4.39.

```
exhale acc(a.f, wildcard)
exhale acc(a.g, wildcard)
```

Figure 4.39: Exhales for field permissions.

In those two `exhale` statements, we lost some unknown amount of permissions on `a.f` and `a.g` and we will never get it back. So we cannot acquire write permissions to `a.f` or `a.g` any more. Hence they cannot be changed. This means that executing those two exhale statements *freezes* the expression `a.f + a.g`. From now on, it will always return the same value, no matter when we execute it.

In the general case, we exhale wildcard permissions on all the fields that appear in the calculation. Since we also want to freeze fields that are used indirectly, i.e. fields that are used by a method or function which is called during the calculation, we also exhale wildcard permissions for fields that appear in access predicates inside preconditions of functions or methods called during the calculation.

Side-effect Freedom

We don't want to restrict the calculation to pure expressions in the SIL sense because we want to allow the creation of new objects, but we don't want the state of of the program to be changed by the calculation.

In order to make sure that the calculation of the lazy val side-effect free even though it is not pure, we have to make sure that no write permissions can be acquired during the calculation. To achieve this, we will replace all permission expressions that appear by wildcard permissions. If only wildcard permissions are present, we can never add permissions together in order to get write permissions, so no side-effects are possible.

4.11.2 Example

An example for a lazy val with a simple calculation that is allowed according to our restrictions can be seen in figure 4.40 on the next page and its translation in figure 4.41 on the following page.

```
lazy val c = a + b
```

Figure 4.40: Example for a lazy val that depends on two fields.

```
c := a + b
exhale acc(a, wildcard)
exhale acc(b, wildcard)
```

Figure 4.41: Translation of the calculation of the lazy val in figure 4.40.

After the evaluation of the lazy val in figure 4.41, the expression on the right hand side cannot change any more.

4.11.3 Controlling The Evaluation

The simplified approach we described before has one drawback. The calculation is evaluated immediately and we have no possibility to delay it. But we might have to initialize certain variables first. We now loosen the constraints slightly. The right hand side doesn't have to be constant from the beginning. But there has to be a certain well-known point such that it becomes constant after that point.

Before the user is allowed to use a lazy val, they have to *commit* it. Committing the value specifies that all the needed values are initialized and the right hand side is ready to be evaluated. It also means that the calculation is freezed, i.e. from now on, the result cannot change any more, no matter when we evaluate it. Before a lazy val is committed, it cannot be used yet.

Committing a lazy val gives away all the permissions needed to calculate it without giving them back and performs the calculation immediately. Since the permissions needed for the calculation are thrown away the right hand side is freezed, so it is guaranteed that the calculation will return the same value in Scala, no matter when it is first evaluated. Usually, lazy vals are just committed in the constructor.

Note that it would be unsound to give back all the permissions since then the values the calculation of the lazy val depends on can change, hence the result of the calculation can change and if the Scala expression is evaluated for the first time after the result has changed, it will get a different result. This results in different semantics in Scala and in SIL.

Delaying The Evaluation

Usually, lazy vals are committed at some point in the constructor if the values used in their calculations are initialized. In some cases, however, it may be necessary to delay the evaluation and only commit after the constructor.

In order to track if a lazy val `b` has been committed already, we introduce an additional assertion `uncommitted(b)`. If the user commits `b` outside the constructor, he has to carry that assertion around via contracts until he commits it.

For the translation of `uncommitted(b)` an additional ghost field `uncommittedB` has to be introduced. We translate `uncommitted(b)` as follows:

```
// Write permissions on the ghost field
acc(uncommittedB, write) &&
  // Not committed
  uncommittedB
  // Write permissions on the lazy val field
  acc(b, write)
```

Figure 4.42: Translation of `uncommitted(b)`.

Commit Example

An example for this idea is given in figure 4.43. The lazy val `b` depends on the mutable field `a` which has to be initialized, first. We commit it in the constructor, which means we throw away read permissions to `a`. After that, we can never get full permissions on `a` again, so we cannot change it anymore. This means that no matter when `b` is actually evaluated in the Scala code, it will always return the same value as it returns if we evaluate it directly. So we can safely evaluate it directly in SIL.

```
lazy val b = a + 4

var a = 3
commit(b)
```

Figure 4.43: Committing a lazy val.

4.11.4 Reading Lazy Vals

An uncommitted lazy val `b` cannot be read yet. We have to make sure that it is committed when it is read by using again the ghost field `uncommittedB` which tracks if `b` is committed. In order to read a committed lazy val `b`, the

user also needs read permissions to it like for any other field. This is very important for us because it is internally translated into a field on which we need to have permissions anyway. An example of a method which reads a lazy val can be seen in figure 4.44 and its translation in figure 4.45.

```
lazy val b = 4 + 5

def foo() = {
  requires(read(b))
  b + 17
}
```

Figure 4.44: Method which reads a lazy val.

```
var b: Int

var uncommittedB: Bool

method foo() returns (result: Int)
  requires acc(b, wildcard)
  requires acc(uncommittedB, wildcard)
  requires !uncommittedB
{
  result := b + 17
}
```

Figure 4.45: Translation of a lazy val read.

More generally, access predicates on a lazy val `b` are always translated as

```
// Read access to the value itself
acc(b, wildcard) //
  // Read access to ghost field
  acc(uncommittedB, wildcard) &&
  // Has been committed
  !uncommittedB
```

Figure 4.46: Translation of `read(b)` (or any other access predicate on `b`).

4.11.5 Calculating The Preconditions

For the committing of a lazy val, we have have to calculate the expression on the right hand side. So we have to fulfill the contracts of all the functions and methods called on the right hand side. We also need all the permissions on fields that are read, all the predicates that appear in unfolding expressions. So we have to extract all those contracts from the right hand side and add wildcard access predicates for all permissions we need etc. We don't allow method calls which require any permissions other than abstract read permissions and wildcard permissions. We also translate abstract read permissions into wildcard permissions. An example for the extraction of contracts from the lazy val defined in figure 4.47 can be seen in figure 4.48.

```
@pure
def f(x: Int) = {
  requires(x > 0)
  x
}

def m(x: Int, y: Boolean) = {
  requires(read(d) && x > 0)
  if (y) x + d else d
}

lazy val b = m(f(a), c)
```

Figure 4.47: A lazy val that makes use of fields, a function and a method call.

```
acc(a, wildcard) && a > 0 && // from f(a)
  acc(c, wildcard) && // from c
  acc(d, wildcard) && f(a) > 0 // from m(...)
```

Figure 4.48: Precondition extracted from the expression of the lazy val in figure 4.47.

4.11.6 Additional Restriction

We decided to allow only one statement with a side effect, namely a method which has to be the outermost expression, i.e. all arguments including the receiver have to be pure. This is necessary to make the calculation of the postcondition possible which will be described in the next section. There is a simple workaround for this restriction, though: The user can put the any calculation into a separate method and add contracts for that method on his own. This shifts the responsibility for inferring the contracts from the

translator to the user. The previous example in figure 4.47 on the previous page fulfills this property, while the examples in figure 4.49 don't.

```
@pure
def f(x: Int) = x
def m() = 0
def n() = 0

// Several side effects are not allowed.
lazy val b = m() + n()

// Method call has to be the outermost expression.
lazy val b = f(m())
lazy val b = m() + 1
```

Figure 4.49: Lazy vals which don't fulfill our restrictions.

4.11.7 Calculating The Postconditions

We also want to compute the postcondition for the calculation of the lazy val automatically. If the calculation is pure, we want to add the property that the lazy val is equal to the right hand side to the postconditions. But this equality alone would lead to problems with the self-framing constraint of postconditions. We need to add the whole precondition we calculated for the expression to the postcondition in order to ensure self-framing.

An example of a lazy val with a pure expression on the right hand side which makes use of a field can be seen in figure 4.50. Its postcondition can be seen in 4.51.

```
var a = 7

lazy val b = a + 5
```

Figure 4.50: A simple lazy val that depends on another field.

```
acc(a, wildcard) && // Precondition, added for self-framing
  b == a + 5 // Equality of the lazy val and the expression
```

Figure 4.51: Postcondition extracted from the expression of the lazy val in figure 4.50.

If a method is called on the right hand side, calculating the postcondition

requires some more work. We cannot state in a postcondition that the lazy val is equal to the result of a method call since only pure expressions are allowed in contracts. However, we know that the method call is the outermost expression. So we can just use the postcondition of that method as the postcondition of the commit method. In the postcondition of the method, we have to replace the `result` variable with the lazy val and the formal arguments with the expressions we used as the arguments of a call. For well-formedness, we again have to add the preconditions of all the arguments to the postcondition, e.g. all access predicates for fields used in arguments. An example of a lazy val that uses a method call can be seen in figure 4.53 and its postcondition can be seen in 4.53.

```
var a = 7
var b = -5

def m(x: Int) = {
  requires(x > 0)
  requires(read(b))
  ensures(read(b))
  ensures(result[Int] > b + x)
  x + b + 10
}

lazy val c = m(a + 5)
```

Figure 4.52: A lazy val that uses a method call.

```
  // Needed for self-framing of the postcondition
  acc(a, wildcard) &&
    // Postcondition of m with a + 5 as the replaced argument
    acc(b, wildcard) &&
    // Postcondition of m with a + 5 as the replaced argument
    // and c as the replaced result
    c > b + (a + 5)
```

Figure 4.53: Translation of the postcondition of the non-pure calculation in figure .

4.12 The Commit Method

The body of the commit method of a lazy val `b` performs the calculation on the right hand side of the lazy val and assigns it to the field of the lazy val. Additionally, it assigns `false` to the ghost field `uncommittedB`.

The precondition consists of the precondition calculated for the calculation of the lazy val and the translated version of `uncommitted(b)`. The latter is needed because it gives us write access predicates to both `b` and `uncommittedB` and this is exactly the reason why the user has to carry around `uncommitted(b)` if he wants to delay the committing.

The precondition consists of the postcondition calculated for the calculation of the lazy val and the translated version of `read(b)`. So the user can use the lazy val the commit statement and he can also assume the postcondition of the calculation.

The commit method for the lazy val seen in figure 4.53 on the previous page can be seen in figure 4.54 as an example.

```
var a: Int
var b: Int
var c: Int
var uncommittedC: Bool

method commitC() returns ()
  // Required for the access on a.
  requires acc(a, wildcard)
  // Preconditions of m
  requires a + 5 > 0
  requires acc(b, wildcard)
  // Standard commit preconditions
  requires acc(c, full)
  requires acc(uncommittedC, full)
  requires uncommittedC
  // Standard commit postconditions
  ensures acc(c, wildcard)
  ensures acc(uncommittedC, wildcard)
  ensures !uncommittedC
  // Needed for well-formedness of a + 5
  ensures acc(a, wildcard)
  // Postconditions of m with the replaced argument
  ensures acc(b, wildcard)
  ensures c > b + (a + 5)
{
  b := a + 5
  uncommittedB := false
}
```

Figure 4.54: Translation of the lazy val in figure 4.53 on the preceding page and its commit method.

4.12.1 Future Work

Because we have observational side-effect freedom on the right hand side of lazy vals, it may be possible to lift some other restrictions. Maybe a method call which is not the outermost expression or even several method calls would be possible.

4.13 Advanced Type System

In this section, we will continue our explanations from 4.3 on page 25 for the more advanced parts of the type system.

4.13.1 Type Parameters of Methods

Another important feature of the Scala language are type parameters. Methods and classes can have type parameters. We will look at type parameters of methods, first. For each type parameter of a Scala method, we introduce an additional parameter of type `ScalaType` for the corresponding SIL method or function and use this parameter to specify the type information in the contracts as described in section 4.3.1 on page 26.

If `A` is one of the type parameters, we can specify the type information of the arguments of type `A` in the contracts, but we still have to give the arguments a SIL type. Should it be `Bool`, `Int` or `Ref`? In Scala, `A` can be instantiated as `Boolean`, `Int` or as any other class, so we have a problem, no matter which one we choose.

A solution would be to use an approach similar to C++ templates. We look at methods with only one type parameter, first. We translate every method with one type parameter three times. Once with `Bool`, once with `Int` and once with `Ref`. Note that we only need the extra parameter of type `ScalaType` and the contracts which introduce the type information for the `Ref` version. So a Scala function as seen in figure 4.55 will be translated as 4.56 on the following page.

```
// Inside class C
@pure
def id[A](a: A): A = a
```

Figure 4.55: Scala method with a type parameter.

```
function id(A: ScalaType, this: Ref, a: Ref): Ref
  requires this != null
  requires isSubType(typeOf(this), C())
  requires isSubType(typeOf(a), A)
  ensures isSubType(typeOf(result), A)
{
  a
}

function idInt(this: Ref, a: Int): Int
  requires this != null
  requires isSubType(typeOf(this), C())
{
  a
}

function idBool(this: Ref, a: Bool): Bool
  requires this != null
  requires isSubType(typeOf(this), C())
{
  a
}
```

Figure 4.56: Simple SIL translation with type parameters as normal arguments, C++ template approach.

The problem with this approach is that it doesn't scale. A with 6 type parameters would already yield $3^6 = 729$ SIL methods. The trick about the C++ templates is, that they only get instantiated if they are actually used. This doesn't work in our case since we also have to verify methods which never get called, e.g. if a library is verified a method that is intended for the user may never get called inside the library. It seems as if it may suffice to verify only one of the 729 versions, but this becomes unclear as soon as the user uses the type information inside the body of the method in a non-trivial way. We will describe a better solution in the next section.

4.13.2 Boxing

A better solution is to do something that is similar to Java boxing. We just wrap primitives in a `Ref`. Since we also want to do that inside pure contexts, we don't create a new object to wrap the primitives and instead introduce a new domain which contains functions to wrap and unwrap plus the appropriate axioms. The full domain can be seen in figure 4.57 on the next page

```
domain BoxingHelper {

  function int2Ref(int: Int): Ref

  function ref2Int(ref: Ref): Int

  function bool2Ref(bool: Bool): Ref

  function ref2Bool(ref: Ref): Bool

  axiom IntTyp {
    forall x: Int ::
    { int2Ref(x) }
    typeOf(int2Ref(x)) == scalaInt()
  }

  axiom IntInverse {
    forall x: Int ::
    { ref2Int(int2Ref(x)) }
    ref2Int(int2Ref(x)) == x
  }

  axiom BoolTyp {
    forall x: Bool ::
    { bool2Ref(x) }
    typeOf(bool2Ref(x)) == scalaBool()
  }

  axiom BoolInverse {
    forall x: Bool ::
    { ref2Bool(bool2Ref(x)) }
    ref2Bool(bool2Ref(x)) == x
  }
}
```

Figure 4.57: The Boxing domain.

Now we use the first version of the translations of `id` in figure 4.56 on the facing page. In a call like in figure 4.58 we just wrap the boxing function around the argument, which results in the translation shown in 4.59 on the following page.

```
assert(id[Int](2) == 2)
```

Figure 4.58: Scala method call with an `Int`.

```
assert(ref2Int(id(Int, this, int2Ref(2))) == 2)
```

Figure 4.59: Translation of a call to a parametrized function with boxing.

Note that in this case, the verifier can use the boxing axioms and the function body to prove that

```
ref2Int(id(Int, this, int2Ref(2))) == ref2Int(int2Ref(2)) == 2.
```

General Case

We will not state the translation rules for methods and functions again, we will just state how they are extended for methods and functions with type parameters. In the general case, i.e. for a Scala method (pure or non-pure) with type parameters $A_1, \ldots, A_n$, we add additional parameters

```
A1: ScalaType, ..., An: ScalaType
```

to the SIL method or function at the beginning of the formal argument list. For each formal argument (or return value) b that has type A_i in the Scala signature, we use `Ref` as the type of the translated formal argument (or return value) b. We then use the local variable A_i which comes as a formal argument to specify in the precondition or postcondition for the return value) of the method or function that

```
isSubType(typeOf(b), A1).
```

4.13.3 Type Parameters of Classes

Type parameters of classes are slightly more complicated. Note that if we have a class `C` with type parameters, it is not possible to just add those type parameters to every method of `C` as we did in section 4.13.1 on page 75 since classes can also have fields whose types depend on type parameters. Since `Cell[A]` may not be the same as `Cell[B]`, it is not sufficient any more to encode `C` as a single constant. We chose to encode it as a type constructor which takes types as an arguments and returns another type. We also add a helper function for each type parameter which can extract that type parameter from a type parameter function application. So for the class in figure 4.60, we would generate the functions and axioms shown in figure 4.61 on the next page.

```
class Cell[A](var a: A)
```

Figure 4.60: A parametrized Scala class.

```
function Cell(A: ScalaType): ScalaType

function CellA(typ: ScalaType): ScalaType

axiom CellAExtraction {
  forall A: ScalaType ::
  { CellA(Cell(A)) }
  CellA(Cell(A)) == A
}
```

Figure 4.61: Functions and axioms for the parametrized type `Cell[A]`.

General Case

For the rest of this section, we will not use the translate function to specify how the types are translated. Actually, this wouldn't even work since we don't want to generate an AST which is added to the main part of the SIL program, instead, we have to add domain functions and axioms to the type domain. So we give an axiom for the most general case instead.

For a class $C[A_1, \ldots, A_n]$, we add the type constructor as seen in figure 4.62 and for each A_i, we define the extractor function and the extractor axiom as seen in figure 4.63.

```
function C(A1: ScalaType, ..., An: ScalaType): ScalaType
```

Figure 4.62: Type constructor for the parametrized type $C[A_1, \ldots, A_n]$.

```
function CAi(typ: ScalaType): ScalaType

axiom CAiExtraction {
  forall A1: ScalaType, ..., An: ScalaType ::
  { CAi(C(A1, ..., An)) }
  CAi(C(A1, ..., An)) == Ai
}
```

Figure 4.63: Extractor function for the type parameter A_i.

4.13.4 Using Parametrized Types

We can now use this to specify that an expression has parametrized types and make the types of fields dependent on the type of their receiver. We can for example consider a simple Scala method as seen in figure 4.64 on the following page.

```
// Inside SomeClass
@pure
def value(cell: Cell[Int]): Int {
  requires(cell ne null)
  requires(read(cell.a))

  cell.a
}
```

Figure 4.64: Pure method which has an argument of a parametrized type.

```
function value(this: Ref, cell: Ref): Int
  requires this != null
  requires isSubType(typeOf(this), SomeClass())
  requires isSubType(typeOf(cell), Cell(Int()))
  requires acc(cell.a, wildcard) &&
    isSubType(typeOf(cell.a), CellA(typeOf(cell)))
{
  ref2Int(cell.a)
}
```

Figure 4.65: Translation of parametrized types and their field types.

Remember that in the example in figure 4.64, according to what we explained in section 4.3.1 on page 26, we have to add contracts which constrain the type of the argument `cell` to the translated SIL function.

Using the type constructor function defined in figure 4.61 on the previous page, we can define the type of the argument `cell` as `Cell(Int())`. We also explained that we translate every access predicate into an access predicate plus the type information. This means that we also have to constrain the type of `cell.a`. We have to use the type of the receiver, in this case `typeOf(cell)` and the type parameter extractor function `CellA` and we get the type `CellA(typeOf(cell))`.

Finally, we need to use the boxing function from section 4.13.2 on page 76 for the body of the function. This gives us the translation shown in figure 4.65.

General Case

We now have to describe how appearances of parametrized types and type parameters of classes are translated. For an appearance of a parametrized type, the type arguments are just translated recursively and then the type constructor is applied to them.

For type parameters of classes, our translation depends on how they are used. There are five situations in which types can appear that are relevant to our translation:

- We inhale the type of a newly created object.
- We inhale the type of the fields of a newly created object.
- We attach type information of fields to access predicates.
- We add type informations to method parameters or return values in contracts.
- They appear as type arguments of other types.

For each of them, we have a different way how a type parameter is treated:

- In the first case, a type parameter can never appear, we cannot create a new object whose type is a type parameter. Things like `new A()` are just not valid Scala code if `A` is a type parameter.
- If we want to inhale that the type of a field of a newly created object `obj` is `T` for some class type parameter `T`, we can use whatever we used as a type argument for `T` when we inhaled the type of `obj`.
- If we attach type information of fields to access predicates, we have to make the type parameter depend on the type of the receiver, i.e. we extract them from the type of the receiver. We want to say that a field access `exp.f` with an arbitrary expression `exp` as the receiver has type `T` for some class type parameter `T`. Let us say that the type parameter extractor function of `T` is `ExtractT`, we then translate `T` as

  ```
  ExtractT(typeOf(exp))
  ```

  in this situation.
- If add type informations to method parameters or return values in contracts, the type parameter of the enclosing class is meant. Since there are no classes in SIL and the only way to get access to the enclosing class is via the `this` parameter, we have to extract the type parameter from that this parameter. We want to say that some parameter (or return value) as type `T` for some class type parameter `T`. Let us say that the type parameter extractor function of `T` is `ExtractT`, we then translate `T` as

  ```
  ExtractT(typeOf(this))
  ```

  in this situation.
- If they appear as type arguments of other types, again the type parameter of the enclosing class is meant. So it is translated analogous to the

previous case. We want to use some class type parameter `T` as a type argument to another type, i.e. as an argument to a type constructor. Let us say that the type parameter extractor function of `T` is `ExtractT`, we then translate `T` as

```
ExtractT(typeOf(this))
```

in this situation.

4.13.5 Type Uniqueness

There is one problem with our approach of encoding parametrized types: Since we don't use constants any more, we cannot use the SIL unique keyword any more. So the verifier doesn't know that the types are really different. It don't even know that `Cell[Int] != Nothing`. This is very annoying since it need this to derive `Null <: Cell[Int]`, which is very important if we want to assign `null` to variables of type `Cell[Int]`. It know that `Cell[Int] <: AnyRef` and it knows that `Null` is a subtype of any subtype of `AnyRef` except for `Nothing`. But here it can't do anything because it might be that `Cell[Int] == Nothing`. We could add an axiom for every pair of types which states that those two types are different. However, we would need $\mathcal{O}(n^2)$ axioms for n types and it gets even worse if we also want him to know that `Cell[Int]` is not the same type as `Cell[Bool]`. If there exists a function f such that

$$f(a) \neq f(b)$$

then $a \neq b$. This follows directly from the well-definedness of functions. The type parameter extractor functions do half the work already since

```
CellA(Cell(Int())) == Int() != Bool() == CellA(Cell(Bool()))
```

gives us

```
CellA(Cell(Int())) != CellA(Cell(Bool()))
```

So we only have to tell the verifier that the types that come from different type constants or type functions are different. We introduce the helper function `typeId` which assigns a unique integer constant to each type through axioms like the ones seen in figure 4.66 on the facing page.

```
axiom NothingTypeId {
  typeId(Nothing()) == 0
}

axiom BarTypeId {
  forall A: ScalaType ::
  { Bar(A, B) }
  typeId(Bar(A, B)) == 1
}
```

Figure 4.66: Axioms to give different types different type ids.

General Case

For all parametrized classes $C[A_1, \ldots, A_n]$ with type parameters $A_1, \ldots, A_n$, we define the type id axiom as seen in figure 4.67 where we replace k with a separate integer for every class in the program.

```
axiom CTypeId {
  forall A1: ScalaType, ..., An: ScalaType ::
  { C(A1, ..., An) }
  typeId(C(A1, ..., An)) == k
}
```

Figure 4.67: Type id axiom for $C[\varphi_1 A_1, \ldots, \varphi_n A_n]$.

4.13.6 Self Subtyping

Subtyping between parametrized types is more complicated than in the case of trivial types since the type parameters of the direct supertype may depend on the type parameters of the subtype. Additionally, type parameters can be invariant, covariant or contravariant and this may result in different subtyping behavior. First we look at subtyping between different instantiations of the same parametrized type. We will call this concept *self subtyping*.

In the case of our cell, the type parameter is invariant, so we have a subtyping relation between two instantiations of the `Cell` type, if and only if they have the same type parameter. This yields the axiom seen in figure 4.68 on the following page. Note that the equivalence operator doesn't exist in SIL, but it is easier to read if we write it down like this.

```
axiom CellSelfSubType {
  forall A: ScalaType, B: ScalaType ::
  { isSubType(Cell(A), Cell(B)) }
  isSubType(Cell(A), Cell(B)) <==> A == B
}
```

Figure 4.68: Axiom for subtyping between different instantiations of `Cell`.

To make matters more interesting, we will introduce another type with a covariant type parameter in figure 4.69.

```
class AbstractCell[+T]
```

Figure 4.69: A class with a covariant type parameter.

Here the self subtyping axiom gets interesting. An instantiation of `AbstractCell` is a subtype of another instantiation if and only if its type parameter is a subtype of the type parameter of the other instantiation. This gives us the axiom seen in figure 4.70.

```
axiom AbstractCellSelfSubType {
  forall A: ScalaType, B: ScalaType ::
  { isSubType(AbstractCell(A), AbstractCell(B)) }
  isSubType(Cell(A), Cell(B)) <==> isSubType(A, B)
}
```

Figure 4.70: Axiom for subtyping between different instantiations of `AbstractCell`.

Contravariance works almost the same way, only the direction of the subtyping of the type parameter has to be changed.

General Case

We define the helper function

`relation(`φ_i`,` A_i`,` B_i`)`

which is

- `isSubType(`A_i`,` B_i`)` if $\varphi_i =$ `"+"`,
- `isSubType(`B_i`,` A_i`)` if $\varphi_i =$ `"-"` and

- A_i == B_i if $\varphi_i =$ "" .

Note that `relation` does not actually appear in the program, we just use it to specify how the axiom is put together.

For all parametrized classes $C[\varphi_1 A_1, \ldots, \varphi_n A_n]$ with variance annotations $\varphi_1, \ldots, \varphi_n$, i.e. $\varphi_i \in \{$ "+" , "-" , "" $\}$ and type parameters $A_1, \ldots, A_n$ and for all type parameters $B_1, \ldots, B_n$ we define the axiom as in figure 4.71. Note that we still have to do the replacements defined by `relation`.

```
axiom CSelfSubType {
  forall A1: ScalaType, ..., An: ScalaType,
         B1: ScalaType, ..., Bn: ScalaType
  { isSubType(C(A1, ..., An), C(B1, ..., Bn)) }
  isSubType(C(A1, ..., An), C(B1, ..., Bn)) <==>
    relation(φ1, A1, B1) &&
      relation(φ2, A2, B2) &&
      ... &&
      relation(φn, An, Bn)
}
```

Figure 4.71: Axiom template for subtyping between different instantiations of $C[\varphi_1 A_1, \ldots, \varphi_n A_n]$.

4.13.7 Type Extractors and Subtyping

Now we will get to subtyping between different parametrized types. We consider the types seen in figure 4.72.

```
class Foo[+A, B]

class Bar[C] extends Foo[C, Int]
```

Figure 4.72: Two parametrized types with a subtyping relation.

The first problem we have here is the following: If a method is defined inside `Foo`, it can use the type parameter `A`. But as we defined it, the extractor function `FooA` only works for `Foo` itself, so if we call this function for an object of type `Bar`, we cannot extract the type parameters. So we need additional axioms for the type parameter extractors on subtypes like the ones shown in figure 4.73 on the next page. Note the difference of the two axioms because `A` is covariant while `B` is invariant. For contravariant type parameters, we define an axiom similar to the one for covariant type parameters, we just replace `isSubType(A(t), A))` by `isSubType(A, A(t)))`

```
axiom FooASubTypeExtraction {
  forall t: ScalaType, A: ScalaType, B: ScalaType ::
  { isSubType(t, Foo(A, B)), FooA(t) }
  isSubType(t, Foo(A, B)) ==> isSubType(FooA(t), A))
}

axiom FooBSubTypeExtraction {
  forall t: ScalaType, A: ScalaType, B: ScalaType ::
  { isSubType(t, Foo(A, B)), FooB(t) }
  isSubType(t, Foo(A, B)) ==> FooB(t) == B
}
```

Figure 4.73: Type extractor axioms for subtypes of `Foo`.

General Case

For the general axiom template, we use `relation` as defined above and for all classes $C[\varphi_1 A_1, \ldots, \varphi_n A_n]$ with variance annotations φ_i and type parameters A_i, we define the axiom we get if we do the replacement for `relation` for the axiom template shown in figure 4.74.

```
axiom CA_iSubTypeExtraction {
  forall t: ScalaType, A_1: ScalaType, ..., A_n: ScalaType ::
  { isSubType(t, C(A_1, ..., A_n)), CA_i(t) }
  isSubType(t, C(A_1, ..., A_n)) ==> relation(φ_i, CA_i(t), A_i)
}
```

Figure 4.74: Axiom template for extractor functions for subtypes.

Subtyping Between Parametrized Classes

The only thing that is still missing is the subtyping between `Foo` and `Bar` itself. This axiom is shown in figure 4.75. Note that we don't need to take the variance of the type parameters into account here because this is already done via the self subtyping axiom.

```
axiom BarSubFoo {
  forall C: ScalaType ::
  { Bar(C) }
  isSubType(Bar(C), Foo(C, Int()))
}
```

Figure 4.75: Subtyping axiom for `Bar[C] <: Foo[C, Int]`.

The verifier can use this subtyping axiom to get the subtyping relations between different parametrized classes and the self subtyping axiom for subtyping relations between different instantiations of the same parametrized class. He can combine the two and use transitivity to derive the subtyping relations if both the classes and the parameters are different.

General Case

For the general rule, assume we the subtyping relation

```
C[φ1A1, ..., φnAn] <: D[B1, ..., Bm]
```

from Scala for some classes C and D. The φ_i are again variance annotations, but they are not used for the axiom. The A_i are the type parameters of C. The B_i are the types which appear as the type arguments of D in the `extends` clause, i.e. if we have

```
class C[φ1A1, ..., φnAn] extends D[Int, Int]
```

we would get B_1 `==` B_2 `==` `Int`. Note that they are not the type parameters of D but rather the instantiations of the type parameters of D in the `extends` clause. However, the B_i can of course be parametrized types again and they can also depend on the A_i. The axiom can be seen in figure 4.76.

```
axiom BarSubFoo {
  forall A1: ScalaType, ..., An: ScalaType ::
  { C(A1, ..., An) }
  isSubType(C(A1, ..., An), D(B1, ..., Bm))
}
```

Figure 4.76: General parametrized subtyping axiom.

4.13.8 Future Work

For full support of the Scala type system, there is still some work to be done. But we are confident that it is relatively easy to extend our existing translation procedures and type system axioms to bounds for type parameters, type members and path dependent types, singleton types, in particular the `this.type` and self-type annotations. Additionally, support for existential types, annotated types, function types and structural types are still missing.

Chapter 5

Implementation

5.1 The Scala Compiler

The main elements of the Scala compiler are trees, symbols and types.

- Trees are used to represent AST nodes, e.g. a variable access or an if expression. A tree usually has a reference to its type and most trees like for example variable accesses also have a reference to the symbol of its declaration. Some trees like `if` expressions don't have a symbol because they don't have a declaration.
- Symbols represent declarations, so for example, every variable declaration, every method and every class gets a symbol. Symbols also have a reference to the type used in the corresponding declaration.
- Types are used to represent the types of expressions or a declarations. Types have a reference to the symbol of their declaration, for example, type `A` has a reference to the symbol of class `A`.

The description of trees, symbols and types can be read in more detail in [11]. In figure 5.2 on the next page, we show the relations between the trees of the terminal nodes of the AST and their symbols and types of the Scala expression in figure 5.1. Note that we only included the trees of the terminal nodes we didn't include the trees of the addition and the multiplication.

```
a + a * b
```

Figure 5.1: A simple Scala expression.

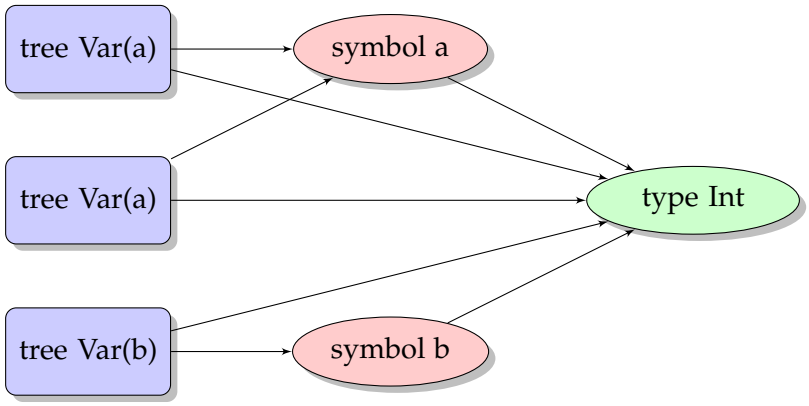

Figure 5.2: Relations between the terminal nodes of the expression in figure 5.1 on the preceding page and their symbols and types.

5.1.1 Compiler Plugins

Extensions of the Scala compiler are usually done via compiler plugins. In order to be able to load and use compiler plugins, the Scala compiler requires the plugin to have a certain structure. A good complete tutorial on how to write a compiler plugin can be found in [1]. Basically, a main class of the plugin has to be defined in an XML file. The plugin class defines the name and a description of the compiler plugin and has the task to create and assemble the components of this plugin and provide them to the compiler. An example implementation of a compiler plugin can be seen in 5.3.

```
class MyPlugin(val global: Global) extends Plugin {
  import global._

  val name = "MyPlugin"
  val description = ""
  val components = List[PluginComponent](new MyComponent)
}
```

Figure 5.3: A simple compiler plugin.

The plugin components do the actual work. They also create a phase which gets an AST from the compiler and can do whatever we want it to do with the AST. They specify a name for this phase and after or before which other phases they have to be run. A simple implementation of a plugin component can be seen in figure 5.4 on the facing page.

[1] http://www.scala-lang.org/node/140

```
class MyComponent(val global: Global) extends PluginComponent {
  import global._

  val runsAfter = List("refchecks")
  val phaseName = "myphase"
  def newPhase(prev: Phase) = new MyPhase(prev)

  class MyPhase(prev: Phase) extends StdPhase(prev) {
    def apply(unit: CompilationUnit) {
      // Do something with the AST in unit.
    }
  }

}
```

Figure 5.4: A simple compiler plugin component.

5.1.2 Path-dependend Types in the Compiler

One difficulty we encountered during our project was the fact that the Scala compiler makes heavy use of path-dependend types [13]. A path-dependend type can be declared as seen in figure 5.5.

```
class A {
  type B
}
```

Figure 5.5: A declaration of a path-dependend type.

We will not go into the details of their usage since we don't need the whole set of features here. There is just one property that is important for us: The type now depends on the instance of A. That is, if we have two instances `a1` and `a2` of `A`, then `a1.B` and `a2.B` are not necessarily compatible, i.e. variables of type `a1.B` might not be assignable to variables of type `a2.B` and vice versa. Even if `a1` and `a2` actually always hold the same object at runtime, they are incompatible if the type system is not able to guarantee that they are actually the same object.

The types of the symbols, trees and types are actually the path dependend types `global.Symbol`, `global.Tree` and `global.Type`. We can avoid having to write `global` all the time by putting `import global._` somewhere. The difficulty of this approach is that it is not as easy to split the translation into several classes which communicate with each other by calling methods and passing around trees, symbols or types. An example of a naive attempt

which is rejected by the compiler can be seen in figure 5.6. Note that this example is of course a simplified setting. In this setting, we could just solve the problem by moving all the methods into the same trait, but as soon as we have a bigger code base and we want to separate the tasks, this approach is not an option any more.

Note that we will use examples that seem similar to our actual work, but the classes and traits used in our examples have nothing to do with our actual project, they just got similar names and are used because we thought that they provide an intuitive insight on the problems of path-dependend types. But the problems and examples are not only relevant for our project, but also for any other Scala compiler plugin which has to work with the AST and analyze it or translate it to something else.

```
trait Translator {
  val global: Global
  val helper: Helper

  def translate(tree: global.Tree) = {
    if (helper.isWhileLoop(tree)) {
      // ...
    } else {
      // ...
    }
  }

}

trait Helper {
  val global: Global

  def isWhileLoop(tree: global.Tree) = {
    // ...
  }
}
```

Figure 5.6: A class which gives a tree to another class via a method argument.

The problem is that the compiler doesn't know that `Translator.this.global` and `Helper.this.global` will be initialized with the same object and hence it will think that `global.Tree` inside `Helper` is not the same type as `global.Tree` inside `Translator`. So the call to `isWhileLoop` will fail.

It is tempting to introduce a type parameter `G <: Global` for the type of `global` both traits and make the field `helper` of `Translator[G <: Global]` have type `Helper[G]`. However, the compiler rejects this as well as using the types

of trees as type parameters.

However, we can solve this problem by surrounding the two traits by another trait as seen in figure 5.7.

```
trait EnclosingTrait {
  val global: Global

  trait Translator {
    val global: EnclosingTrait.this.global.type =
      EnclosingTrait.this.global
    val helper: Helper

    def translate(tree: global.Tree) = {
      if (helper.isWhileLoop(tree)) {
        // ...
      } else {
        // ...
      }
    }

  }

  trait Helper {
    val global: EnclosingTrait.this.global.type =
      EnclosingTrait.this.global

    def isWhileLoop(tree: global.Tree) = {
      // ...
    }
  }
}
```

Figure 5.7: An enclosing trait is added such that they have the same `global`.

The problem is now solved by telling the compiler that the `global` used by both `Translator` and `Helper` is the same object as the one from `EnclosingTrait`. So now the compiler knows that we are dealing with the same `Tree` type in both classes and the method call succeeds.

5.1.3 Splitting the Code

However, if we only had this approach at our disposal, we would have to put all the classes and traits of our translation into one single trait which would lead to a single huge file. As long as the dependencies between our classes are acyclic, we can just make the enclosing traits extend each other

in a way that for each class or trait, the used classes or traits are available. Still the fact that the enclosing traits are in a subtyping relation and hence use the same `global` and the inner classes use the same `global` as well makes sure that the compile allows us to pass around symbols, types and trees. An example of this approach can be seen in figure 5.8.

```
trait TranslatorComponent extends HelperComponent {

  trait Translator {
    val global: TranslatorComponent.this.global.type =
      EnclosingTrait.this.global
    val helper: Helper

    def translate(tree: global.Tree) = {
      if (helper.isWhileLoop(tree)) {
        // ...
      } else {
        // ...
      }
    }

  }
}

trait HelperComponent {
  val global: Global

  trait Helper {
    val global: HelperComponent.this.global.type =
      EnclosingTrait.this.global

    def isWhileLoop(tree: global.Tree) = {
      // ...
    }
  }
}
```

Figure 5.8: The traits are splitted, but subtyping still guarantees them to have the same `global`.

At some point, we would have to add a class which mixes in all the traits of our translation, defines the `global` variable to be the one we get from the compiler plugin. This class will be used by the translation plugin component and has a method to run the translation.

With this approach, we can split the different classes into different files and

they are still able to communicate via method calls. However, if we look at the last example, one might think that we could achieve the same thing in a much simpler way. It seems unclear why we even need the inner classes. We could just put all the methods into the outer classes instead and we would achieve the same result. So why don't we do something like the example in figure 5.9 which seems much simpler and cleaner than the previous version?

```
trait TranslatorComponent extends HelperComponent {

  def translate(tree: global.Tree) = {
    if (isWhileLoop(tree)) {
      // ...
    } else {
      // ...
    }
  }

}

trait HelperComponent {
  val global: Global

  def isWhileLoop(tree: global.Tree) = {
    // ...
  }

}
```

Figure 5.9: A seemingly simpler approach to achieve the same thing.

This is indeed a valid approach. We even used it in our project for some very simple utility methods which should be globally available like the translation from Scala source file positions to SIL positions. However, while this approach works well for small examples and very simple situations, it highly limits our options. Basically, we would just get a huge namespace containing all the methods which would behave just like global procedures. We would end up with a setting similar to procedural programming and we would give up all the benefits of object oriented programming. Imagine if we had an interface for `Helper` and several possible implementations. Something like this would not be possible with the approach seen in figure 5.9, so we have to use the seemingly more complicated approach in figure 5.8 on the preceding page.

5.1.4 The Cake Pattern

We still need a solution for cyclic dependencies, though. Luckily, Scala provides something that does the job, namely self-type annotations. A self-type annotation in a trait or abstract class `T` can be used to tell the compiler that any class or trait which inherits `T` has to be a subtype of some given type `A`. The compiler then checks this property for all classes or traits which extend `T`. Inside `T`, it can be assumed that `this` is a subtype of type `A` and in particular, every field and method of `A` can be assumed to be present on `this` as well. An example can be seen in 5.10.

```
trait A {
  def m() {}
  var a = 3
}

trait T {
  // Self-type annotation which requires any
  // class or trait which extends T to be a
  // subtype of A.
  this: A =>

  def foo() {
    // this.a and this.m() can be used because
    // of the self-type annotation.
    m()
    a = 15
  }
}

// This class would be rejected by the compiler.
// It extends T, but it is not a subtype of A.
class Invalid extends T

// This class would be accepted by the compiler.
// It extends T and it is also a subtype of A.
class Valid extends T with A
```

Figure 5.10: A seemingly simpler approach to achieve the same thing.

Using those self-type annotations, we can construct our plugin according to the cake pattern [14]. The idea and the reason for the name is that each component is a piece of the cake and requires via self-type annotations, that it is part of the big cake. The cake is represented by another trait which extends all the components. This way, every component can use the functionality of any other component. Figure 5.11 on the next page shows how we could

apply the cake pattern to our example to allow cyclic dependencies between the components.

```
trait Cake
  extends TranslatorComponent
  with HelperComponent {
  val global: Global
}

trait TranslatorComponent {
  this: Cake =>

  // The self type annotation combined
  // with the fact that Cake extends HelperComponent
  // ensures that we can use HelperComponent here.
  trait Translator {
    val global: TranslatorComponent.this.global.type =
      EnclosingTrait.this.global
    val helper: Helper

    def translate(tree: global.Tree) = {
      if (helper.isWhileLoop(tree)) {
        // ...
      } else {
        // ...
      }
    }
  }
}

trait HelperComponent {
  this: Cake =>

  // The self type annotation combined
  // with the fact that Cake extends TranslationComponent
  // ensures that we can use TranslationComponent here.
  trait Helper {
    val global: HelperComponent.this.global.type =
      EnclosingTrait.this.global

    def isWhileLoop(tree: global.Tree) = {
      // ...
    }
  }
}
```

Figure 5.11: Using the cake pattern to allow cyclic dependencies between components.

The cake pattern allows us to use the compiler and its symbols, types and trees in a typesafe way without using casts while still splitting our code into different files and without giving up the benefits of object-oriented programming.

5.2 The Scala2Sil Compiler Plugin

The Scala2Sil compiler plugin consists of the components

- `WellDefinednessComponent`,
- `TypeCollectionComponent`,
- `SubTypeCollectionComponent`,
- `SignatureCollectionComponent`,
- `ContractsTranslationComponent`,
- `TranslationComponent`,
- `OutputComponent` and
- `VerificationComponent`.

They correspond to the phases explained in section 3 on page 15. Each component defines an operation which is applied to all the files of the program being compiled. We are forced to use several plugin components since some components need the previous component to be run on all files before it can be run. For example, we need the signatures of all files before we can start translating the contracts since a function which is defined in another file might be called inside a contract. We could merge some of them, though, for example the output component and the verification component could be merged into one component. Since we need several components anyway, it is conceptually cleaner to separate those tasks.

5.2.1 Communication Between the Components

The downside of using several plugin components is that we have to manage the communication between the components somehow. They are activated by the Scala compiler and since the compiler doesn't give us the possibility to hand informations to the next component, we have to do something on our own. We do have the control over the construction of the components, though. So if two components have to communicate, we give a shared object to both of them in the constructor. The component which runs first can store something into the fields of that object, the component which runs later can then use those stored informations. We used this pattern in the following cases:

- The `WellDefinednessComponent` passes a list of errors to the `TypeCollectionComponent`.
- The next components of the list use a different way to communicate as we will soon explain, but they all forward the errors and if the error list is not empty, they all do nothing.
- The `TranslationComponent` passes a complete SIL program or a list of errors to the `OutputComponent`.
- The `OutputComponent` outputs the program into a file or as debug output, if no errors are found and if a file path is given or if the verbose flag is set.
- The `OutputComponent` just forwards its input, a SIL program or a list of errors to the `VerificationComponent`.
- The verification component passes the SIL program it got to the verifier, if the list of errors it was given is empty.

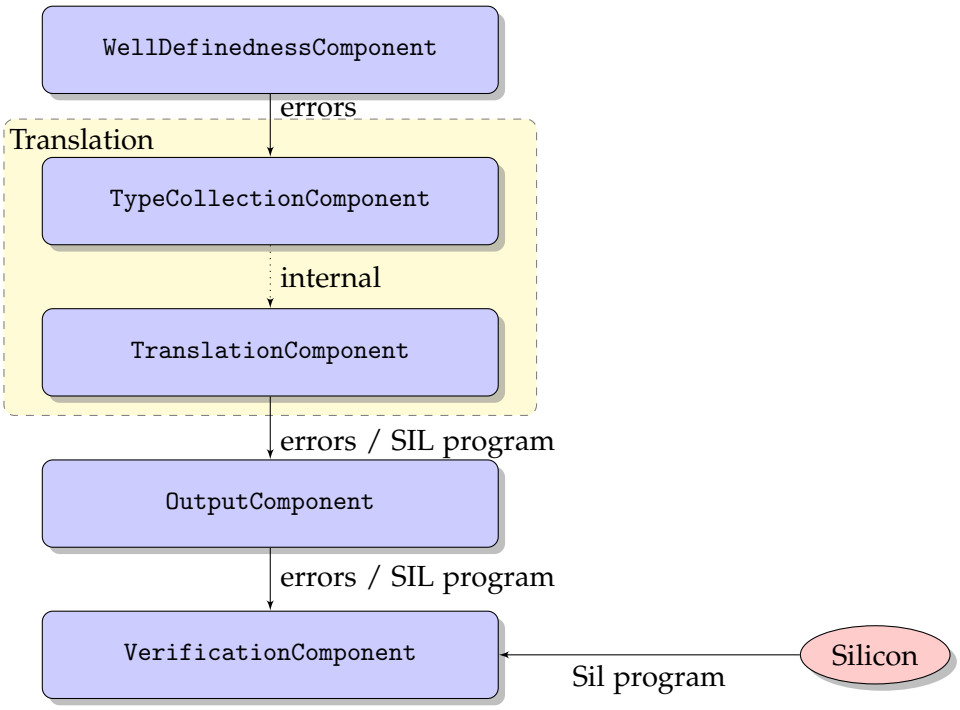

Figure 5.12: Communication between phases.

It gets more complicated if the information which has to be passed around contain symbols or other path dependent types that depend on `global`. The type system has to be able to guarantee that they all use the same `global`, so just using different classes which take `global` as an argument doesn't suffice. Type parameters don't help here, yet, since path dependent types didn't

seem to be supported as type parameters at the time we made this decision. So we put the phases

- `TypeCollectionComponent`,
- `SubTypeCollectionComponent`,
- `SignatureCollectionComponent`,
- `ContractsTranslationComponent` and
- `TranslationComponent`

as inner classes into a trait `TranslationComponents` and they all use the same `global` instance which is a field of this enclosing trait. This pattern is also used in the Scala compiler itself.

The enclosing trait `TranslationComponents` also has fields for the types, fields, predicates, functions and methods etc. which get filled in by the phases. Those fields are basically used for the communication between the phases. The type collection phase fills in the types, the subtype collection component fills in some more types and adds some axioms for the relations between them, the signature collection components fills in the signatures for fields, predicates, functions and methods, the contracts translation component adds contracts to them and finally, the translation phase adds the bodies and assembles the final SIL program.

5.3 The SIL Name Generator

One problem we had to solve for our translator was the naming of variables, functions, fields, predicates and methods. In Scala, it is possible that two classes in different package or two fields in different classes have the same name. A Scala identifier might even collide with the name of an internal function or axiom. In SIL, we have a global namespace except for local variables or variables bound by a quantifier, so this is not possible. Building huge prefixes which include all enclosing packages, classes or methods would help to some extend, but they would make the SIL code unreadable, so we opt for something more elegant. Only in the case of methods, fields, functions and predicates, we add the class name only as a prefix since this enhances readability. Additionally, Scala allows almost any Unicode sign in identifiers while SIL is very restrictive at this point and only allows alphanumeric symbols. Also, a name like `unique` is allowed as a variable name in Scala, while in SIL, this is not a valid identifier since it is a reserved word.

5.3.1 Generating Identifiers

Generating valid identifiers from arbitrary strings works as follows: If the input string is empty, we return an arbitrary letter, `v` in our case. If the

first letter of the input string is valid at any later position in a SIL identifier, but not as the first character, we prepend an arbitrary letter, again `v` in our implementation. Then we go through all the characters of the input string and if we encounter a character which is not valid inside a SIL identifier we look in a fixed look-up table if we have a valid replacement for this character. If yes, we replace it accordingly, otherwise we delete it. Those replacements are very useful if the input string contains mathematical operators or Greek letters. For example `+` is replaced by `plus` which is still recognizable for any reader of the SIL program.

5.3.2 Rendering Identifiers Unique

The simplest way of making identifiers unique is to store all generated identifiers and if we generate a new one, we check if that one already exists. If so, we append a serial number at the end. While this pragmatic approach definitely works, it will render the SIL code quite ugly since we would get a lot of unnecessary numbers after almost every variable. For example, the `this` variable would get a different number in every method. Method names should still be globally unique, but local variables of different methods should be allowed to get the same names. However, a local variable should still not collide with a global name.

We solved this by creating tree of name contexts such that every name context can create and store identifiers which are unique among the identifiers of all its ancestors and descendants. However, two contexts for which none is an ancestor of the other may generate the same identifier. We implemented this as a SIL extension such that other projects can make use of it as well. But this means that it has to be thread safe. Our current projects may not make use of multithreading yet, but since this might be the case for future projects, SIL itself has to be thread safe. So we have to take care of locking and deadlock freeness as well.

At the beginning, we have one global context with id (0), a lock, an empty subcontext list and an empty map from identifiers to numbers. A context supports the following operations:

Create Subcontext Let l be the number of subcontexts of the current context before the operation. We create a new subcontext which gets the id of the current context with l appended as its id, a lock, an empty subcontext list and an empty map from identifiers to numbers. We append the new context to the subcontext list of the current context and return the new context.

Render Identifier Unique Let i be the input identifier. Let r be the relevant contexts, i.e. all ancestors and descendants of the current context. We acquire the locks of all contexts in r in the lexicographic order of their

ids. If the identifier map of no context inside r contains i, we add the mapping $i \to 0$ to the identifier map of the current context and i is the result. If some identifier map contains i, let m be the maximum value i maps to in any of the identifier maps or r. Let i_m be the string we get if we concatenate the identifier i with _ and a string representation of m. While i_m exists in any of the identifier maps of r, increase m by one. Put $(i \to m)$ and $(i\_m \to 0)$ into the identifier map of the current context and i_m is the result. We release the locks in the lexicographic order of the ids and return the result.

Since we add every identifier we return to the map and only return identifiers that have not been in the map before, this fulfills the property that no conflicting identifiers will be generated in two contexts where one of them is an ancestor of the other. Keeping an identifier map instead of a set is just a performance enhancement. If the same identifier is used as input very often, the algorithm doesn't have to count from the start to the current number but it can jump to the current number directly. Deadlock freeness is guaranteed by the total ordering of the ids and because we acquire the locks in the order of the ids. Note that we only introduced the ids to make the deadlock freeness more clear, they don't really have to be stored since ordering the relevant nodes by their ids yields the same result as just doing Breadth-first search starting with the root context on the relevant contexts.

Chapter 6

Conclusion

We successfully implemented a tool that translates Scala to SIL which supports a substantial amount of features including classes, the type system, pattern matching, expressions with side effects and full Implicit Dynamic Frames support.

We also found an interesting and powerful way to encode Scala lazy vals to SIL which enables the user to use them with minimum extra effort in the most common cases.

Reviewing our goals from section 1.1 on page 2, we can state that we successfully reused the experience the group has gained on Chalice on Scala, solved several new problems and provided a solid basis for future projects.

6.1 Future Work

One possible future extensions is the support for inheritance. Note that there exists a recent project that introduces inheritance to Chalice [4] and we are confident that the solutions from that work could be applied to Scala as well. Until now, no libraries are supported yet, however, this could be achieved without any changes to the current project if contracts for the Scala libraries are provided. Additionally, support for termination checking, concurrency, objects, i.e., global lazy vals, higher order functions could be added.

Appendix A

The API of our Library

- `@pure` Annotation to mark a method as a pure function.
- `@predicate` Annotation to mark a method as a predicate.
- `requires(e)` Precondition. Only allowed at the beginning of methods. A precondition can be specified as an argument. That argument has to be pure. Has no runtime behavior.
- `ensures(e)` Postcondition. Only allowed at the beginning of methods. A postcondition can be specified as an argument. That argument has to be pure. Has no runtime behavior.
- `invariant(e)` Invariant. Only allowed at the beginning of while loops. An invariant can be specified as an argument. That argument has to be pure. Has no runtime behavior.
- `assert(e)` Assertion. Allowed in an arbitrary position inside a method. An assertion that has to be proved can be specified. That argument has to be pure. Has no runtime behavior.
- `assume(e)` Assumption. Allowed in an arbitrary position inside a method. An assumption for the verifier can be specified. That argument has to be pure. Has no runtime behavior.
- `old[T](e)` Old Expression. Only allowed in preconditions of methods. An expression can be specified as an argument and the old expression stands for the value of that expression before the execution of the method. Has no runtime behavior.
- `result[T]` Result expression. Only allowed in postconditions. Stands for the result of the method. Has no runtime behavior.
- `read(f)` Read access predicate. Only allowed in positive positions of an expression, that is, only in conjuncts or on the right hand side of an

implication. Only allowed in contracts, invariants, assertions, assumptions, fold statements, unfold statements and unfolding expressions. For the latter three cases, only a predicate access is allowed as an argument, in the other cases, also a field access is possible. Can be used to specify the presence of read permissions on a predicate or a field. Has no runtime behavior.

- `write(f)` Write access predicate. Only allowed in positive positions of an expression, that is, only in conjuncts or on the right hand side of an implication. Only allowed in contracts, invariants, assertions, assumptions, fold statements, unfold statements and unfolding expressions. For the latter three cases, only a predicate access is allowed as an argument, in the other cases, also a field access is possible. Can be used to specify the presence of write permissions on a predicate or a field. Has no runtime behavior.
- `acc(f, p)` General access predicate. Only allowed in positive positions of an expression, that is, only in conjuncts or on the right hand side of an implication. Only allowed in contracts, invariants, assertions, assumptions, fold statements, unfold statements and unfolding expressions. For the latter three cases, only a predicate access is allowed as the first argument, in the other cases, also a field access is possible. The second argument has to be a permission expression consisting of the special permission values, floats, multiplication, subtraction and addition. Can be used to specify the presence of a certain amount of permissions on a predicate or a field. Has no runtime behavior.
- `rd` Abstract read permission. Only allowed in permission expressions. Represents an arbitrary permission value p with $0 < p \leq 1$ which is guaranteed to be the same in every occurrence inside a method. Has no runtime behavior.
- `rd_!` Global read permissions. Only allowed in permission expressions. Represents an arbitrary permission value p with $0 < p \leq 1$ which is guaranteed to be the same throughout the whole program. Has no runtime behavior.
- `fold(write(P))` Fold statement. Takes an access predicate on a predicate as an argument. Unfolds this predicate. Only allowed in non-pure contexts. Has no runtime behavior.
- `unfold(write(P))` Unfold statement. Takes an access predicate on a predicate as an argument. Unfolds this predicate. Only allowed in non-pure contexts. Has no runtime behavior.
- `unfolding(write(P)) { foo() }` Unfolding expression. Takes an access predicate on a predicate as its first argument and an expression as the second argument. Unfolds this predicate. Allowed in any context

where the expression in the second argument is allowed. Evaluates and returns the expression in the second argument at runtime.

- `commit(a)` Commit statement. Takes a lazy val as its argument. Commits this lazy val, see section 4.11 on page 66. Only allowed in non-pure contexts. Has no runtime behavior.
- `uncommit(a)` Uncommitted expression. Takes a lazy val as its argument. Can be used to specify that the lazy val in the argument has not been committed yet, see section 4.11 on page 66. Only allowed in positive positions of an expression, that is, only in conjuncts or on the right hand side of an implication. Only allowed in contracts, invariants, assertions and assumptions. fold statements, unfold statements and unfolding expressions. Has no runtime behavior.
- `a ==> b` Implication. Can be used in any context that allows boolean expressions. Behaves like implication at runtime.

Bibliography

[1] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: An overview. pages 49–69. Springer, 2004.

[2] John Boyland. Checking interference with fractional permissions. In *Proceedings of the 10th international conference on Static analysis*, SAS'03, pages 55–72, Berlin, Heidelberg, 2003. Springer-Verlag.

[3] Andreas Bühlmann. Permission-based verifcation of subclassing and traits. Master's thesis, ETH Zurich, January 2013.

[4] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical report, J. ACM, 2003.

[5] Stefan Heule, K. Rustan M. Leino, Alexander J. Summers, and Peter Müller. Fractional permissions without the fractions. In *IN FTFFP*, 2011.

[6] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: a powerful, sound, predictable, fast verifier for c and java. In *Proceedings of the Third international conference on NASA Formal methods*, NFM'11, pages 41–55, Berlin, Heidelberg, 2011. Springer-Verlag.

[7] K. Rustan M. Leino. This is Boogie 2. Manuscript KRML 178, 2008. Available at http://research.microsoft.com/ leino/papers.html.

[8] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *LPAR (Dakar)*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[9] K. Rustan M. Leino and Peter Müller. A basis for verifying multi-threaded programs. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009*, volume 5502 of *Lecture Notes in Computer Science*, pages 378–393, Berlin, March 2009. Springer-Verlag.

[10] Rokas Matulis. Extensible code contracts for scala. Master's thesis, ETH Zurich, September 2012.

[11] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An introduction to Scala. August 2005.

[12] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Proc. ECOOP'03*, Springer LNCS, 2003.

[13] Martin Odersky and Matthias Zenger. Scalable component abstractions, 2005.

[14] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. *ACM SIGPLAN Notices*, 40(1):247–258, January 2005.

[15] Malte Schwerhoff. Symbolic execution for chalice. Master's thesis, ETH Zurich, 2011.

[16] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 148–172, Berlin, Heidelberg, 2009. Springer-Verlag.

[17] Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. Satisfiability modulo recursive programs. In *Proceedings of the 18th international conference on Static analysis*, SAS'11, pages 298–315, Berlin, Heidelberg, 2011. Springer-Verlag.