Gallimaufry : An Automated Framework for Proving Type Soundness

Anne Mulhern

May 17, 2005

Chapter 1

Introduction

1.1 Motivation and Goals

For many years, type-safety has been considered an essential property of a language. Type-safety was an important goal for the designers of the Java language and is one of its primary selling points. Generally, Java programmers can expect to find many of their bugs revealed as type errors at compile time; in contrast, C programmers experience such bugs as inexplicable behavior or crashes at runtime. Java is also considered to be far more secure because of its type-safety. Even the initial version of Java was quite complicated, and no formal proof was given for its type-safety. Proofs of Java's type-safety were subsequently developed using a variety of techniques [1, 3, 11, 12, 17, 25, 31, 32], and Java is now generally considered to have been proven type-safe.

Around any reasonably popular language there will arise a large population of variants. These variants may arise to address limitations in the original language's expressiveness or to redress flaws in its design. Java is an outstanding example. There were many contenders for a version of generics for Java; a version based on GJ [15] has now been incorporated into Java 1.5 (Tiger) [19]. At the time that GJ was designed, it was not demonstrated to be type-safe. A proof of the type-safety of GJ followed only subsequently, and, in fact, work on the proof exposed at least one bug in the original design of GJ [18].

Although type-safety is a desirable property of a language, considerations of type-safety continue to be of secondary importance. One reason is historical; the designers of FORTRAN and others of the earlier high level language were more interested in abstracting the basic operations of the underlying machine for the convenience of the programmer than type-safety. Another is cultural; the benefits of type-safety are not well understood outside the programming languages community. Proofs of type-safety are difficult to understand and construct and real languages are complex; this makes proving type-safety for a real language a considerable undertaking.

Gallimaufry [22] is a new kind of framework for experimenting with language extensions. The novel aspect of this framework is the support that it provides for proving type-soundness of language extensions. The proof is based on a translational semantics developed in detail by Kim Bruce in "Foundations of Object-oriented Languages: Types and Semantics" [5]. The semantics defines a translation from a sim-

ple object-oriented language, $\mathcal{SOOL}$, to a variant of the lambda calculus which is itself type-sound. A user can experiment with language extensions by defining additional syntax and translation rules and can rely on Gallimaufry to verify the type-soundness of the modified language.

A dictionary gives the meaning of gallimaufry variously as

1. A hash of various kinds of meat, a ragout.
2. Any absurd medley; a hodgepodge.

It is derived from the French *gallimafrée* meaning sauce or ragout. Many feel that certain programming languages have become a gallimaufry in the sense of the second definition and have echoed the protest made by the poet Edmund Spenser in the 16th century:

> So now they have made our English tongue a gallimaufry, or hodgepodge of all other speeches.

Extending a language with some part of another language can be achieved successfully, but when done injudiciously may lead to awkward retroactive patches and far-reaching difficulties as illustrated by the following example.

1.2 Example: Covariant Subtyping of Arrays in Java

Java allows covariant subtyping of arrays. This means that if A is a subtype of B, the Java compiler can infer that A[] is a subtype of B[] and consequently allow assignment of a B[] array to an A[] array reference. This particular subtyping rule is expressed formally in the following fashion.

$$\frac{A <: B}{A[] <: B[]}$$

Figure 1.1 shows an unfortunate consequence of this rule. The effect of the first two lines is to make two aliases, `sa` and `oa` to a single `String` array. At the third line, an `Integer` object is substituted for the `String` object at index 0 of the array. And at the fourth line, the method `charAt()` is invoked on the element at index 0, which is now an `Integer` object. Unfortunately, the method `charAt()`, although defined for `String` objects, is not defined for `Integer` objects. This code passes the Java type checker but is clearly not safe.

In early versions of Java, execution of the code did not halt until somewhat after line 3. Later, this was recognized as unacceptable and a dynamic type check was added at each store to an array. In the current version of Java, execution halts at line 3 with an `ArrayStoreException`.

The root of the problem is the covariant subtyping of arrays which allows the assignment at line 2 to occur. The purpose of Gallimaufry is to alert users to the parts of a type system which make a programming

```
1)  String[] sa = new String[]"zero";
2)  Object[] oa = sa;
3)  oa[0] = new Integer(0);
4)  sa[0].charAt(0);
```

Figure 1.1: Example: Covariant Subtyping of Arrays in Java

language unsound. In this case, Gallimaufry would automatically identify that the covariant subtyping rule made Java unsafe.

Covariant subtyping of arrays in Java continues to be a hindrance as enhancements to the language are developed. It has greatly muddled implementation of Java Generics because of the interaction of type erasure and runtime checking. It has forced the language designers to treat arrays and classes defined using generics very differently, even though superficially they may appear quite similar and as if they should be used in consistent ways. It is not clear whether the designers of Java were unaware that covariant subtyping for arrays made the language unsafe or whether they discounted the importance of this problem. In any case, in hindsight, it is clear that any benefit gained from the increase in expressiveness from covariant subtyping of arrays is far outweighed by the cost in complexity of the Java Generics enhancement and the necessity of executing a runtime check when an element is inserted into an array.

1.3 Definitions

We give a few brief definitions of terms, following Cardelli in [8]. A *trapped* error is an error which causes execution of the program to stop immediately. A good example of such an error is dividing by zero or an `ArrayOutOfBoundsException` in Java. An *untrapped* error is an error which allows execution to continue. C programmers are familiar with this kind of error. It generally results in a core dump, but often at an instruction far removed from the original cause of the error. A *safe* language is a language which allows no *untrapped* errors.

Forbidden errors should include all *untrapped* errors as well as a selection of *trapped* errors. What is considered a forbidden error may vary from language to language. *Well behaved* programs are programs that have no forbidden errors. A *strongly checked* language allows no forbidden errors. Strong checking implies safety. A *weakly* checked language is one in which some unsafe programs are allowed, i.e., forbidden errors do not include all untrapped errors.

Types are used to designate the set of possible values a variable can contain. A *typechecker* is used to check the types of a program. A *well typed* program is one that passes the typechecker. A programming language is *type sound* if every well-typed program, i.e., every program that passes the typechecker, is well-behaved, i.e., allows no forbidden errors

Reconsider the example in Figure 1.1. One can deduce that early versions of Java were not safe, since insertion of an `Integer` into a `String` array was *untrapped*. Adding the dynamic check that occurs at line 3 removes this cause of unsafety, since this error is now *trapped*. However, the current version of Java is still not type sound since the assignment at line 2, which necessitates the dynamic check at line 3 is still permitted.

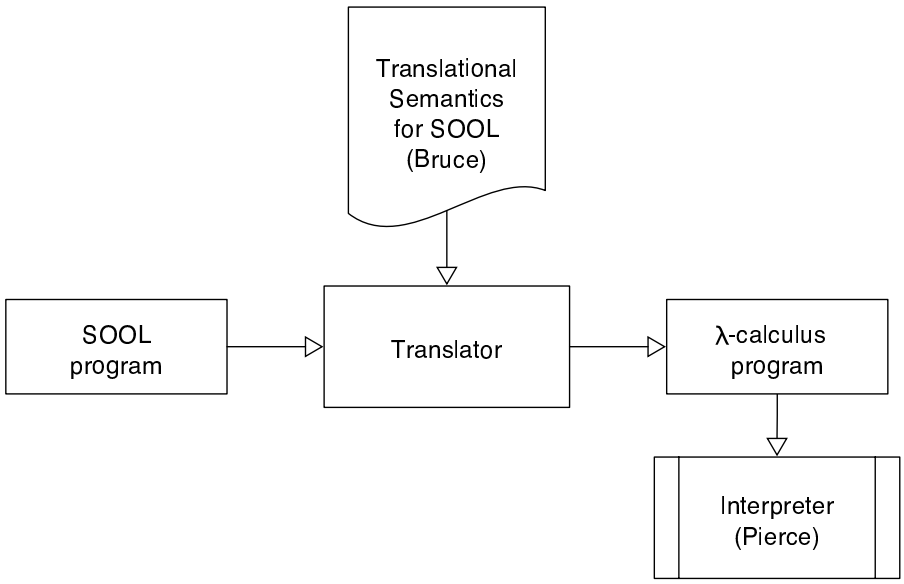

Figure 1.2: The translator transforms a $\mathcal{SOOL}$ program to a lambda calculus program.

The standard example of a type sound language is ML. Languages that are safe are Java and Lisp. Lisp is entirely dynamically checked, while Java achieves safety through a combination of static and dynamic checks. C is not safe.

Gallimaufry checks type soundness of a language. This might seem to prevent its use with languages that incorporate some dynamic checks to insure safety, but that is not the case. One way that Gallimaufry can be used with languages that have a dynamic type-safety component is to restrict Gallimaufry to checking only the parts of the language that are intended to be type sound. Another possibility is to modify a programming language so that expressions that are dynamically type-checked are transformed to expressions that can be statically type-checked. For example, cast expressions, which might fail at runtime, can be made into guarded cast expressions, where the guard guarantees that the cast will never fail at runtime.

1.4 Overview of Gallimaufry

The core of Gallimaufry is a translator from $\mathcal{SOOL}$(the simple object-oriented language of [5]) to a variant of the lambda calculus. The translator is written in O'Caml. We have chosen as a target for the translation a variant of the lambda calculus described by Pierce in [28] for which an interpreter has been developed [27]. Thus Gallimaufry provides a "reality check"; a program written in the extended language can be translated to a lambda calculus and executed using an existing interpreter. Figure 1.2 depicts the translator.

A proof of type soundness is constructed using the Coq automated proof assistant [10]. Coq proof obligations are generated by hand from the descriptions in [5]. Coq structures are generated from the code of the translator. The whole is then used by Coq to generate the proof. Figure 1.3 depicts the prover. The translator on the far left is retained from Figure 1.2; the code of the translator is transformed into Coq structures. Note the Coq tactics that are also generated by hand. These are hints to the Coq proof assistant that guide it in constructing the proof of type-safety.

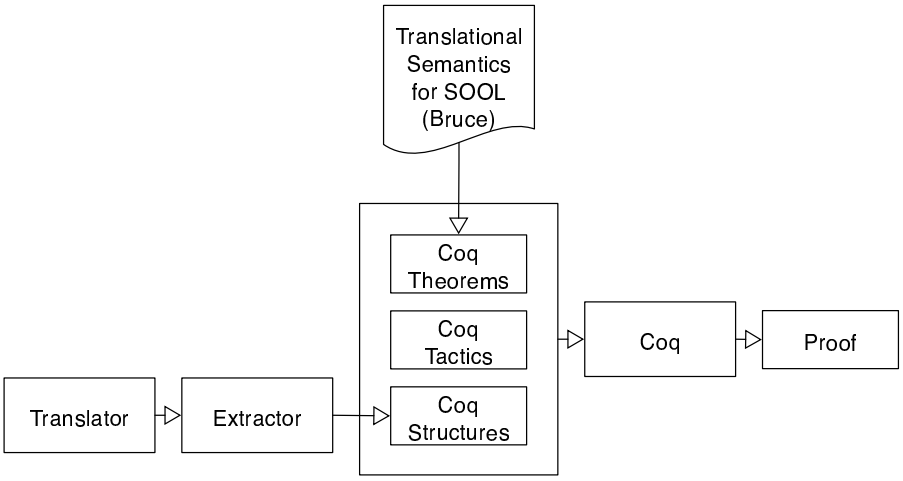

Figure 1.3: The prover generates a proof of type-safety for the $\mathcal{SOOL}$ language.

A user of Gallimaufry specifies a language extension by specifying additional syntax and translation rules. The user specifications are encoded in the source code of the translator. Gallimaufry responds by generating a new proof of type soundness for the enhanced language or generating an error message. In order to generate the new proof, Gallimaufry must regenerate the Coq structures derived from the translator and, in general, the Coq tactics as well.

1.5 Outline

In the following chapter we discuss some of the theoretical and technical principles underlying Gallimaufry. The first section outlines the general requirements for development of the proof. The next section discusses the $\mathcal{SOOL}$ language itself, including important differences between it and Java, and how typechecking is done in the language. In the next section we discuss how to take the typechecking rules from the previous section and express them; first in the translator (as O'Caml datatypes and functions) and then again in the prover (as Coq structures). This section serves as an example demonstrating how the rules of the $\mathcal{SOOL}$ language can be transformed relatively straightforwardly into Coq structures. In the next section we give a brief discussion of the translational semantics for $\mathcal{SOOL}$. In the final section we demonstrate how we use Coq to prove a property of the translation. We choose a simplified example: proving that subtypes are preserved by the translation for a subset of the types of the $\mathcal{SOOL}$ language.

In the next chapter we summarize the contributions of Gallimaufry.

In the final chapter we describe a tentative schedule.

Chapter 2

Automatically Proving Type Safety Using a Translational Semantics

2.1 Overview

Gallimaufry uses an indirect method to prove the type-safety of $\mathcal{SOOL}$. The semantics of $\mathcal{SOOL}$ is given via a translation to a variant of the lambda calculus which is itself believed to be type sound. If it can be proved that every well-typed program in $\mathcal{SOOL}$ is translated into a well-typed program in the lambda calculus then it can be inferred that $\mathcal{SOOL}$ is type sound. It is not necessary to make any statements about the case where a $\mathcal{SOOL}$ program does not typecheck.

Proving type soundness using a translational semantics requires developing a translation from the source language, the language for which type soundness is to be proved, to the target language, the language for which type soundness is proved, and demonstrating that the translation is correct, i.e., sound and well defined. The translation is sound if the translation of every well-typed source language expression is well-typed in the target language and if the translation of the type of the source expression is the type of the translated expression. The translation is well defined if there exists a unique translation for every expression.

2.1.1 Preservation of Types

A correct translation requires that types be preserved. In Figure 2.1 we see this property illustrated; if we move to the right and then down, i.e., find a type for the $\mathcal{SOOL}$ expression and then translate that type to a lambda calculus type, or if we move down and then to the right, i.e., translate the $\mathcal{SOOL}$ expression to a lambda calculus expression and then find its type the resulting types must be identical.

Actually, the translator incorporates the $\mathcal{SOOL}$ typechecker, and a $\mathcal{SOOL}$ expression cannot be translated if the $\mathcal{SOOL}$ typechecker fails to typecheck it. Thus, in the case where the typechecker cannot give a type to a $\mathcal{SOOL}$ program the translator will not give a translation to the $\mathcal{SOOL}$ expression.

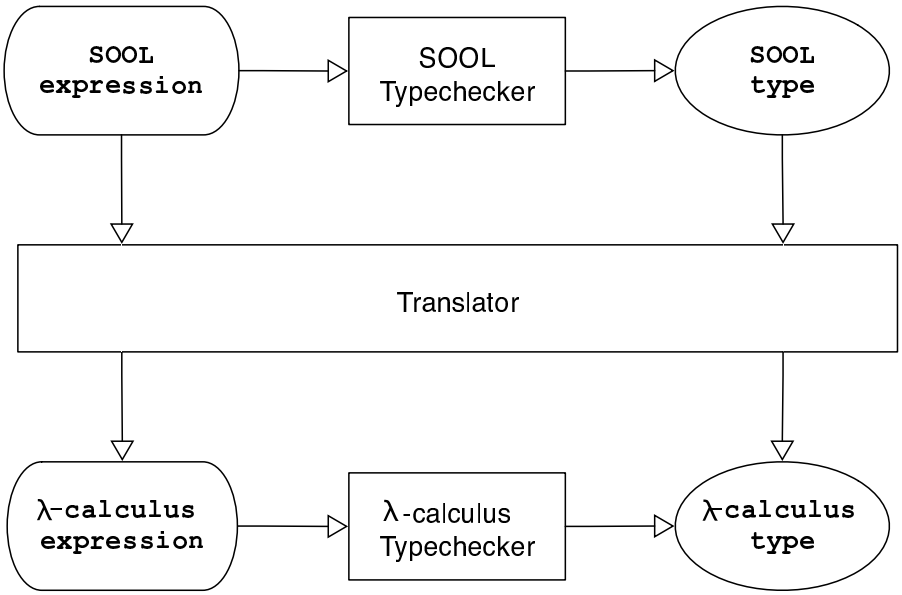

Figure 2.1: The types of expressions must be preserved by the translation.

Preservation of Subtypes

If the source language incorporates a notion of subtyping it is necessary to demonstrate that subtypes are preserved by the translation in order to demonstrate that types are preserved.

In Figure 2.2 we see this property illustrated; if two $\mathcal{SOOL}$ types are in the subtype relation then the translations of their types must also be in the subtype relation. This property involves only types and their translations and has nothing to do with the typing of expressions.

In the next section we discuss the $\mathcal{SOOL}$ language in greater detail.

2.2 The $\mathcal{SOOL}$ language

2.2.1 Description of $\mathcal{SOOL}$

The $\mathcal{SOOL}$ language, developed by Bruce in [5], is a simple class-based object-oriented language with first class classes. $\mathcal{SOOL}$ functions and references should be very familiar to anyone with experience using ML. $\mathcal{SOOL}$'s objects and classes may be less familiar; we focus on these in the following discussion.

Refer to Appendix A for a complete description of the syntax of the language.

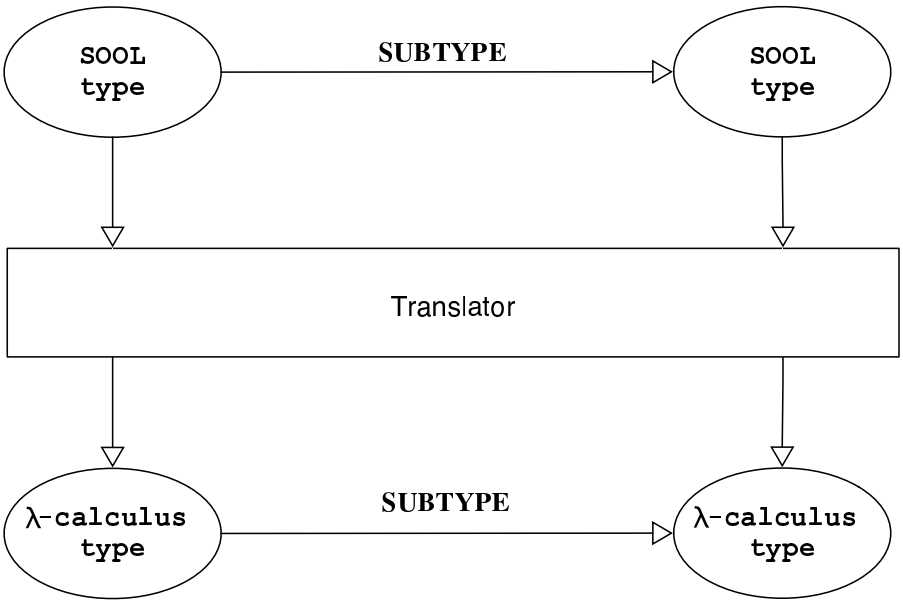

Figure 2.2: Subtypes must be preserved by translation.

Types

Types give an abstract specification of a program's behavior. Types are used to specify many kinds of program behavior, including safety properties [20] and sequential behavior [24]. The most usual and best established use of types is to specify the public interfaces of functions, records, objects, classes, and so forth. This work deals with these types. The underlying theory of such types is developed in [5, 8, 28] and others.

$\mathcal{SOOL}$ type constants are analogous to the primitive types of ML. Type identifiers are names which can be translated to types using $\mathcal{SOOL}$*'s type constraint system*.

Record types are not part of the $\mathcal{SOOL}$ language per se; however, they are the building blocks of object types and class types. A record type is a set of names, where each name has an associated type.

Separation of interface and implementation. In Java, classes define the implementation of an object as well as its type. However, there is a trend among members of the Java development community [2, 16] to separate interface from implementation. Developers define an interface, i.e., the *type* of an object, and may develop multiple implementations of that interface with different properties. The Java Collection Framework, which has at least five distinct implementations of the `List` interface, is a familiar example. This separation, which is a popular programmer idiom [14] in Java, is an integral part of the $\mathcal{SOOL}$ language. A $\mathcal{SOOL}$ `ObjectType` is analogous to a Java *interface* as it gives only the types of the methods of an object. A $\mathcal{SOOL}$ `ClassType` has no analogy in Java. However, the type of a class and the type of an object created using that class must be related as discussed in Section 2.2.2.

`this` or `self`. $\mathcal{SOOL}$ uses the word `self` in the same sense as `this` is used in Java. `VisObjectType` is like an `ObjectType` except that the types of instance fields as well as the types of methods are exposed. In

typechecking the methods of a class, the fields as well as the methods of `self` may be referred to; thus it is necessary to type `self` with `VisObjectType`. This is the only use of `VisObjectType`.

See Appendix A.1 for the complete syntax of $\mathcal{SOOL}$ types.

Class-based Object-oriented Languages

An object-oriented language is a language that uses *object*s to encapsulate data and the methods that may operate on the data. Some languages, such as C++ or Java, provide extensive support for objects but permit programs to be written without the use of objects. A purely object-oriented language is Smalltalk [30], where "everything is an object".

Class-based object-oriented languages use *classes*, which are templates that describe how objects are constructed. Popular examples of class-based object-oriented languages are Java and C++. Prototype-based languages use *prototyping* to generate new objects directly from old objects. One example of such a language is the research language Self [29]; a better known example is ECMAscript [13] which is more commonly known as Javascript.

In most class-based languages, the type of an object, i.e., the types of its field and methods, cannot be changed after it is created while in prototype-based languages such changes are commonly made. The ability to dynamically change the type of an object makes typing considerably more complicated [33].

First Class Classes

In $\mathcal{SOOL}$ classes are first class. This means that they can be assigned to variables and passed as arguments to functions. In contrast, classes are not first class in Java and can be manipulated only through the reflection mechanism.

$\mathcal{SOOL}$ Expressions

Every $\mathcal{SOOL}$ program contains a block, which consists of a (possibly empty) list of type declarations, a (possibly empty) list of constant declarations, and a sequence of statements terminated by an expression. This final expression is the value returned by the program. The reader should not be misled into underestimating the complexity of a $\mathcal{SOOL}$ program. The list of constant declarations may contain declarations of classes, and classes contain declarations of functions. The body of every function is itself a block, so every function may contain new type declarations and new constant declarations, and so on. A $\mathcal{SOOL}$ program can simulate a Java package, for example, by including only constant and type definitions. Figure 2.3 shows an example $\mathcal{SOOL}$ program. The example shows two type declarations, the first an `ObjectType` and the second a `ClassType` declaration, two constant declarations, the first a `class` declaration and the second a reference to an object, and, at the end, the program body.

The program body contains three lines of code. The first line instantiates an object using the `Point` class

and assigns it to the `pt` variable declared in the constant section. The second line invokes the `move()` method on the object, and the third line does nothing. `move()` may be invoked on `pt` because `pt` has type `Ref PointType` which contains the method `move()`. The assignment in the first line requires a formal relationship between the type of the class `Point`, which is `PtClassType`, and the type of `pt`, which is `PointType`. This relationship is discussed in the next section.

See Appendix A.2 for the complete syntax of $\mathcal{SOOL}$ expressions.

2.2.2 Typechecking $\mathcal{SOOL}$ Programs

A *type constraint system*, C, which stores type definitions and a *type environment*, $\mathcal{E}$, which stores the types of expression identifiers are required to typecheck a $\mathcal{SOOL}$ program. Type definitions are added to the type constraint system when they are processed and type bindings for expression identifiers are added to the type environment when constant definitions are processed.

Types and constants declared within a block are not visible outside the block. During typechecking, the type environment and type constraint system may expand as a block is processed, but will contract again when the block is exited.

The Type Constraint System and the Type Environment

The type constraint system, C, is a mapping from type identifiers to type expressions. The function $C(\mathtt{T})$ returns the type expression obtained by expanding the type expression `T` fully according to the mappings in C. The function is defined recursively on the structure of the type expressions.

When adding a type definition to the type constraint systems an occurs check must be used to ensure that $C(\mathtt{T})$ will always terminate, i.e., that none of the types are recursively defined and that type identifiers are not shadowed by new definitions in inner blocks.

See Appendix A.3 for the complete definition.

The type environment, $\mathcal{E}$, is a set of bindings between expression identifiers and type expressions.

Typechecking Definitions

Typechecking type definitions is relatively straightforward. As each type declaration is processed, the corresponding type identifier to type expression mapping is added to the type constraint system. If the type constraint system's occurs check fails then the type definition cannot be typed.

On the other hand, when typechecking constant definitions, the type constraint system is unchanged, but a type binding is added to the type environment when each constant declaration is processed. The type definition fails to type if the constant expression does not type check or if a binding already exists for the

```
Program PointExample;
type
    PointType = ObjectType {|
        move:  Integer × Integer → Command;
        getx:  Void → Integer;
        gety:  Void → Integer
    |};
    PtClassType = ClassType ( {|
        x:  Integer;
        y:  Integer
    |}, {|
        move:  Integer × Integer → Command;
        getx:  Void → Integer;
        gety:  Void → Integer
    |}) ;
const
    Point:  PtClassType = class ( {|
        x:  Integer = 0;
        y:  Integer = 0;
    |}, {|
        move:  Integer × Integer → Command =
            function(dx:  Integer, dy:Integer):  Command is {
                self.x := (val self.x) + dx;
                self.y := (val self.y) + dy;
                return nop } ;
        getx:  Void → Integer = function() :  Integer is {
            return val self.x; }
        gety:  Void → Integer = function() :  Integer is {
            return val self.y } ;
    |}) ;
    pt:  Ref PointType = ref nil;
{
    pt := new Point;
    pt ⇐ move (3, 2);
    return nop
}
```

Figure 2.3: An example $\mathcal{SOOL}$ program

expression identifier. This second condition prevents shadowing of the expression identifier by another constant definition in an inner block.

In the preceding example (see Figure 2.3) the definitions of the types `PointType` and `PtClassType` cause two constraints to be added to the type constraint system, while the definitions of the constants `Point` and `pt` cause two bindings to be added to the type environment.

Type Rules for Expressions and Statements

The typechecking rules of $\mathcal{SOOL}$ are examined in this section.

Most of the rules are straightforwardly defined in terms of their components. Recollect that record types are not part of the $\mathcal{SOOL}$ language. However, since they are used to form other types in the language a rule for typing them is necessary.

`new` followed by a class expression yields an object where the types of the methods of the object are the same as the types of the methods of the class, but the types of fields are not included. This is the typechecking rule that links the type of a class and the type of objects created from the class.

Well-typed statements have type `Command`. The rules for typechecking statements are quite straightforward. Rule *Program* is simply the rule that allows one to typecheck an entire program by typechecking its top-level block.

See Appendix A.4 for a complete list of $\mathcal{SOOL}$ type rules.

Inversion

The typing rules for $\mathcal{SOOL}$ do not in themselves furnish an algorithm for typechecking a $\mathcal{SOOL}$ program. In general, given a set of type rules for a language, it may be possible to type the same expression in two distinct ways. Consequently, by applying the type rules in one way, the typechecker may succeed in typechecking the program, and by applying the type rules in another way, may reject the program as not well typed.

We want to make the more powerful statement that if an expression of a certain form is typable at all, then it has a unique type. This is known as the *inversion* property for a given language and type system.

For example, for the $\mathcal{SOOL}$ language, we want to make the following statements about the expressions `false` and `true`.

- If `true` can be typed at all, then it has type `Boolean`.
- If `false` can be typed at all, then it has type `Boolean`.

Or, more formally,

- If $\mathcal{C}, \mathcal{E} \vdash$ `true` : `R`, then $\mathcal{C}, \mathcal{E} \vdash$ `R = Boolean`.
- If $\mathcal{C}, \mathcal{E} \vdash$ `false` : `R`, then $\mathcal{C}, \mathcal{E} \vdash$ `R = Boolean`.

For `val` and `ref` expressions, which are recursively defined in terms of their subexpressions, we would like to be able to show that the types of the expressions have a certain form, recursively defined in terms of the types of the subexpressions.

- If `val E` can be typed at all, then `E` has type `Ref T` and `val E` has type `T`.
- If `ref E` can be typed at all, then `E` has type `T` and `ref E` has type `Ref T`.

Or, more formally,

- If $\mathcal{C}, \mathcal{E} \vdash$ `val E` : `R`, then $\mathcal{C}, \mathcal{E} \vdash$ `E` : `Ref T` and $\mathcal{C}, \mathcal{E} \vdash$ `R = T`.
- If $\mathcal{C}, \mathcal{E} \vdash$ `ref E`: `R`, then $\mathcal{C}, \mathcal{E} \vdash$ `E` : `T` and $\mathcal{C}, \mathcal{E} \vdash$ `R = Ref T`.

The inversion lemma is sometimes known as the *generation* lemma, since, given a valid typing statement it shows how a proof of this statement could have been generated. Practically, the inversion lemma can be transformed into a recursive algorithm for calculating the type of a term, since it defines, for each syntactic form, how to calculate its type.

Thus, if we can make statements like those above for *every* syntactic form defined in the language then we can mechanically generate a typechecker for the language.

In the next section we discuss the Gallimaufry typechecker for the simple version of the $\mathcal{SOOL}$ language described so far.

2.3 The Gallimaufry Typechecker for $\mathcal{SOOL}$

The following discussion illustrates how the $\mathcal{SOOL}$ language and $\mathcal{SOOL}$ typechecking rules are incorporated into Gallimaufry. We show first how the language and typechecker are expressed in the translator and then discuss the prover. To simplify the discussion the examples use only a fragment of the $\mathcal{SOOL}$ language.

2.3.1 Overview

A component of the Gallimaufry translator is the typechecker for $\mathcal{SOOL}$. Such a typechecker is necessary, as the translation of $\mathcal{SOOL}$ programs is not defined unless they are well typed.

```
let rec et exp = match exp with
  | Sool.ExpId(_) -> None
  | Sool.ExpConstant(exp') -> type_constexp exp'
  | Sool.ExpValue(exp') ->
    (match et exp' with
       | (Some (Sool.TyReference(t''))) -> Some(t'')
       | _ -> None
    )
  | Sool.ExpReference(exp') ->
    (match et exp' with
      | Some(t') -> (Some (Sool.TyReference(t')))
      | _ -> None
    )
  | ...
```

Figure 2.4: Example: Typechecking $\mathcal{SOOL}$ Expressions.

The translator is written in O'Caml [26], a variant of the ML language. The types and expression of $\mathcal{SOOL}$ are defined in the natural way using algebraic types. The inversion lemma allows us to express the typechecking of expressions in a straightforward manner. Figure 2.4 shows a portion of the typechecker implementation. The typechecker returns an *option* type; in the case where an expression cannot be typechecked it returns the value `None`. For the `Sool.ExpValue` case it is necessary to check whether the type of the subexpression is `Ref`. This matches exactly the condition required by the typing rule in the previous section. For the `Sool.ExpReference` case no such check is required; the subexpression is typechecked and if the check is succesful the expression itself can be typechecked.

In this example, the typechecker returns `None` for the `Sool.ExpId` case. This is an appropriate action when no binding for the identifier exists in the type environment. We defer discussion of the type environment and type constraint system until after discussion of the prover, as the constraints imposed by the prover affect our treatment of these data structures in the translator.

2.3.2 Expressing the Typechecker in Coq

Expressing the algebraic datatypes used to represent the syntactic forms of $\mathcal{SOOL}$ is extraordinarily straightforward. Figure 2.5 shows an *Inductive Definition* for a subset of $\mathcal{SOOL}$ expressions on the right and an O'Caml algebraic datatype for the same subset. The correspondence is obvious.

Expressing the typechecker itself however presents considerably more difficulty. The natural way appears to be to express the recursively defined function as a Coq *Fixpoint* definition but such an approach results in a number of difficulties as discussed below.

The type environment and type constraint system. As mentioned previously, the type constraint system and type environment are augmented when the initial type definitions and constant definitions in a block are processed. When typechecking expression identifiers the type environment must be checked for a binding

```
type exp =
  | ExpId of string
  | ExpConstant of constexp
  | ExpValue of exp
  | ExpReference of exp
  | ...
```

O'Caml algebraic datatype

```
Inductive exp :  Set :=
  | ExpId :  String.string -> exp
  | ExpConstant :  constexp -> exp
  | ExpValue :  exp -> exp
  | ExpReference :  exp -> exp
  | ...
```

Coq Inductive Definition

Figure 2.5: Example: $\mathcal{SOOL}$ Expressions in O'Caml and Coq

for the identifier. In the previous examples, we have always assumed that the type environment was empty; hence no expression identifier can be typed successfully.

The most obvious way of including the type environment and type constraint system in the typechecker is to pass them as additional parameters in the form, for example, of associative lists. The `Sool.ExpId` case is changed so that the type to which the identifier is bound in the type environment is expanded by the type constraint system. Unfortunately, this simple approach does not work. The flaw in the approach results from the definition of the type constraint systems lookup function.

$C(T)$, i.e., lookup of types in the type constraint system, is recursively defined in terms of the structure of `T` in all cases except where `T` is a type identifier. In that case, the definition is as follows:

If `t` is a type identifier, then $C(\texttt{t}) = C(\texttt{T})$ if $\{ \texttt{t} = \texttt{T} \} \in C$, otherwise $C(\texttt{t}) = \texttt{t}$.

Coq will not accept a Fixpoint definition unless the size of the input to the function decreases with every recursive call. Coq requires that the parameter on which the function decreases be specified in the case where there is more than one parameter to the function. This is a fundamental restriction of Coq Fixpoints. Unfortunately, the `lookup` function must decrease in not one but *two* parameters, i.e, both the type being expanded and the associative list containing the mappings from type identifiers to type expressions.

To solve this difficulty, we decided on a purely functional representation of the type constraint system. We define the type constraint system as a function from types to types; when a new constraint is added to the system a new function is formed from the current function. Figure 2.6 shows partial versions of the method that creates the empty type constraint system, and the one that adds a new binding to the type constraint system. The definition `create` simply creates an anonymous fixpoint structure which operates recursively on recursively defined types, but as the identity on type identifiers. The definition `add` takes a type constraint system parameter and returns a new fixpoint structure which operates recursively on all recursively defined types but invokes the type constraint system parameter in the `Sool.TyId` case.

This purely functional approach is quite straightforward in O'Caml.

We express the type enviroment in this purely functional manner as well. Since the type environment is a straightforward mapping of expression identifiers to types this is not strictly necessary. However, since we intend to make the translation from O'Caml to Coq purely automatic we feel that a uniform approach is desirable.

```
Definition create (tt:unit) := fix l (ty:Sool.ty) : Sool.ty := match ty with
  | Sool.TyId _ => ty
  | Sool.TyConstant _ => ty
  | Sool.TyReference v => Sool.TyReference (l v)
  | ...

Definition add (tcs: Sool.ty -> Sool.ty) (idtype : Sool.idtype) (ty :Sool.ty) :=
fix tcs' (ty':Sool.ty) : Sool.ty := match ty' with
  | Sool.TyId name => match idtype with
    Sool.IdType name' => if (String.equal_bool name name')
      then tcs ty
      else tcs ty'
    end
  | Sool.TyConstant _ => ty'
  | Sool.TyReference v => Sool.TyReference(tcs' v)
  | ...
```

Figure 2.6: Example: Coq Purely Functional Definition for $\mathcal{SOOL}$ Type Constraint System

We have shown how we can represent the typechecker using Coq structures. The techniques described here can be extended to other portions of Gallimaufry.

Now we proceed to discuss the translation of $\mathcal{SOOL}$ programs to λ calculus programs.

2.4 The Translational Semantics

2.4.1 The Target Lambda Calculus

The target for the translation is the polymorphic lambda calculus with subtyping, $\Lambda^{P}_{<:}$ [5, 28]. This calculus is very expressive. It includes

- the fix operator to express recursion,
- existential types and operators for information hiding,
- universal types to express quantification over types, especially parametric polymorphism,
- subtypes,
- records, and
- references

Refer to Appendix B for the complete syntax and typechecking rules of the calculus.

2.4.2 Translation

The semantics of $\mathcal{SOOL}$ is defined by a translation to $\Lambda^P_{<:}$. Types and expressions are defined by translation to appropriate types and expressions of $\Lambda^P_{<:}$.

Refer to Appendix C for the complete translation rules.

Translating Types

The translation of object types yields a pair corresponding to the instance variables and the method of the object. For `VisObjectType` the instance variables are exposed in the translation; for `ObjectType` the instance variables are hidden using an existentially quantified type.

Translating Expressions

The translation of $\mathcal{SOOL}$ expressions depends on the typing of the expression as well as the expression itself. For this reason, the type *judgement* for an expression is used as an argument to the translation function, rather than just the expression. The typechecking rule for functions, for example, requires that a function body be typechecked using a type environment that has been augmented with bindings for each of the function parameters; consequently, the translation rule requires that the function body be translated using the augmented type environment.

2.5 Proving Properties of the Translation

As an example, we show how we prove that subtypes are preserved by translation, or more formally: if $C \vdash \mathtt{S} <: \mathtt{T}$ then $\mathcal{T}_C[\![C]\!] \vdash \mathcal{T}_C[\![\mathtt{S}]\!] <: \mathcal{T}_C[\![\mathtt{T}]\!]$.

For simplicity we restrict the discussion to $\mathcal{SOOL}$ constant types, reference types, and function types.

Refer to Appendix A.4 for the complete $\mathcal{SOOL}$ subtyping rules and to Appendix B for the $\Lambda^P_{<:}$ subtyping rules.

2.5.1 Structure of the Proof

The proof is given by induction on the size of the proof. That is, we assume that for all proofs of length less than n, if $C \vdash \mathtt{S} <: \mathtt{T}$ then there is a proof that $\mathcal{T}_C[\![C]\!] \vdash \mathcal{T}_C[\![\mathtt{S}]\!] <: \mathcal{T}_C[\![\mathtt{T}]\!]$. This assumption is used wherever the conclusion of a subtyping rule includes a subtyping assumption in the premise.

```
Module Typetranslator.

Fixpoint translate (ty : Sool.ty) {struct ty} : Lambda.ty := match ty with
  | Sool.TyConstant const =>
    let (k, c) := Typeconstantmap.translate const in Lambda.TyConstant k c
  | Sool.TyReference v => Lambda.TyReference (translate v)
  | Sool.TyFunction pt rt => Lambda.TyArrow (translate pt) (translate rt)
  .
  .
  .
end.

End Typetranslator.
```

Figure 2.7: Gallimaufry translator expressed as Coq Fixpoint

2.5.2 Coq Structures

In order to construct the proof we need to represent the type translator function and the functions which decide the subtyping relation in $\mathcal{SOOL}$ and in $\Lambda^{P}_{<:}$ using Coq structures.

The type translator function is a $\mathcal{SOOL}$ Fixpoint declaration. See Figure 2.7 for a representation of the translator. Note that the translator uses a constant map function (not shown) to translate $\mathcal{SOOL}$ constant types to $\Lambda^{P}_{<:}$ constant types.

The subtype function takes two types and returns true if the subtype relation holds, otherwise false. In Coq, we represent this function as an *Inductive Definition* (Figure 2.8). It would be possible to represent the subtype functions as Coq Fixpoints, but for technical reasons this technique is preferable. We are dealing with a subset of $\mathcal{SOOL}$ types and the subset of $\Lambda^{P}_{<:}$ types that the translation yields; the only applicable rules for either language are $Reflex_{<:}$ and $Function_{<:}$. In the figure, line (1) encodes the $Reflex_{<:}$ rule and line (2) encodes the $Function_{<:}$ rule for $\mathcal{SOOL}$ and for $\Lambda^{P}_{<:}$. Note that line (2) encodes the *contravariance* rule for function parameter types and the *covariance* rule for function return types discussed in [8].

The subtype preservation rule is stated in Coq thus:

```
forall (ty1 ty2 : Sool.ty),
  Subtype.subtype (ty1) (ty2) ->
    Lambdasubtype.subtype (Typetranslator.translate ty1) (Typetranslator.translate ty2).
```

We give Coq the necessary tactics (Figure 2.9) and Coq generates a proof.

```
Module Subtype.

Inductive subtype : Sool.ty -> Sool.ty -> Prop :=
(1)  | subtype_all: forall (ty1 ty2: Sool.ty), ty1 = ty2 -> subtype ty1 ty2
(2)  | subtype_TyFunction :
           forall (pt1 pt2 : Sool.ty) (rt1 rt2 : Sool.ty),
             subtype rt1 rt2 ->
               subtype pt2 pt1 ->
                 subtype (Sool.TyFunction pt1 rt1) (Sool.TyFunction pt2 rt2)
  .
  .
  .
.

End Subtype.

Module Lambdasubtype.

Inductive subtype : Lambda.ty -> Lambda.ty -> Prop :=
(1)  | subtype_all : forall (ty1 ty2: Lambda.ty), ty1 = ty2 -> subtype ty1 ty2
(2)  | subtype_TyArrow :
           forall (l1 l2 r1 r2 : Lambda.ty),
             (subtype l2 l1) ->
               (subtype r1 r2) ->
                  subtype (Lambda.TyArrow l1 r1) (Lambda.TyArrow l2 r2)
  .
  .
  .
.

End Lambdasubtype.
```

Figure 2.8: $\mathcal{SOOL}$ and $\Lambda^{P}_{<:}$ subtyping rules expressed as Coq Inductive Definitions

```
Proof.
intro ty1.
intro ty2.
intro H.
induction H;
[
  rewrite H; apply Lambdasubtype.subtype_all |

  unfold Typetranslator.translate;  apply Lambdasubtype.subtype_TyArrow
];
auto.
Qed.
```

Figure 2.9: Coq tactics

2.5.3 Deriving Tactics

For the following discussion we switch to a symbolic representation of the hypotheses and goals that we set out to prove. We use the symbols $<:_{\mathcal{SOOL}}$ and $<:_{\Lambda^P_{<:}}$ to represent the subtype inductive definitions of $\mathcal{SOOL}$ and $\Lambda^P_{<:}$ respectively.

We restate the goal symbolically: $\forall \texttt{ty1}\ \texttt{ty2},\ \texttt{ty1} <:_{\mathcal{SOOL}} \texttt{ty2} \Longrightarrow \mathcal{T}_C[\![\texttt{ty1}]\!] <:_{\Lambda^P_{<:}} \mathcal{T}_C[\![\texttt{ty1}]\!]$

The introductory tactics, i.e., `intro`s and `induction` are derived from the structure of the goal itself. Each use of the `intro` tactic makes the variable introduced a hypothesis while simultaneously removing it from the goal. Specifically, `ty1`, `ty2`, and our hypothesis, $\texttt{ty1} <:_{\mathcal{SOOL}} \texttt{ty2}$ are placed in the hypothesis section of the proof, and the remaining goal is just $\mathcal{T}_C[\![\texttt{ty1}]\!] <:_{\Lambda^P_{<:}} \mathcal{T}_C[\![\texttt{ty1}]\!]$.

The induction tactic allows us to split the hypothesis $\texttt{ty1} <:_{\mathcal{SOOL}} \texttt{ty2}$ so that we have two separate goals, each goal corresponding to the rule for the final step in the derivation of the subtype relation for a $\mathcal{SOOL}$ program. Thus, for the current example, the induction steps gives us two goals (see Figure 2.10).

Note that the induction step is quite powerful. Without any prompting Coq yields the two powerful hypotheses $\mathcal{T}_C[\![\texttt{rt1}]\!] <:_{\Lambda^P_{<:}} \mathcal{T}_C[\![\texttt{rt2}]\!]$ and $\mathcal{T}_C[\![\texttt{pt2}]\!] <:_{\Lambda^P_{<:}} \mathcal{T}_C[\![\texttt{pt1}]\!]$ that we need to prove the second goal.

To prove the first subgoal, we simply note that `ty1 = ty2`. This allows us to rewrite the first goal as $\mathcal{T}_C[\![\texttt{ty1}]\!] <:_{\Lambda^P_{<:}} \mathcal{T}_C[\![\texttt{ty1}]\!]$ and the goal is then easily proved using the $Reflex_{<:}$ rule for $\Lambda^P_{<:}$, i.e., $ty1 = ty2 \Longrightarrow ty1 <:_{\Lambda^P_{<:}} ty2$.

Proving the second subgoal is more difficult. We transform the subgoal by applying the translator function, yielding $\mathcal{T}_C[\![\texttt{pt1}]\!] \to \mathcal{T}_C[\![\texttt{rt1}]\!] <:_{\Lambda^P_{<:}} \mathcal{T}_C[\![\texttt{pt2}]\!] \to \mathcal{T}_C[\![\texttt{rt2}]\!]$ for our new subgoal. We can then apply the $Function_{<:}$ rule for $\Lambda^P_{<:}$, i.e., $pt2 <:_{\Lambda^P_{<:}} pt1 \Longrightarrow rt1 <:_{\Lambda^P_{<:}} rt2 \Longrightarrow pt1 \to rt1 <:_{\Lambda^P_{<:}} pt2 \to rt2$. The two implications of the $Function_{<:}$ result in the current goal being replaced by two subgoals, one for the parameter type and one for the return type (see Figure 2.10). These subgoals can be proved automatically by Coq.

Tactics	Proof
intros	$\dfrac{\texttt{ty1} <:_{\mathcal{SOOL}} \texttt{ty2}}{\mathcal{T}_C[\![\texttt{ty1}]\!] <:_{\Lambda^P_{<:}} \mathcal{T}_C[\![\texttt{ty2}]\!]}$
induction	*Reflex*$_{<:}$ subgoal: $\dfrac{\texttt{ty1} = \texttt{ty2}}{\mathcal{T}_C[\![\texttt{ty1}]\!] <:_{\Lambda^P_{<:}} \mathcal{T}_C[\![\texttt{ty2}]\!]}$ *Function*$_{<:}$ subgoal: $\dfrac{\begin{array}{c}\texttt{rt1} <:_{\mathcal{SOOL}} \texttt{rt2} \\ \texttt{pt2} <:_{\mathcal{SOOL}} \texttt{pt1} \\ \mathcal{T}_C[\![\texttt{rt1}]\!] <:_{\Lambda^P_{<:}} \mathcal{T}_C[\![\texttt{rt2}]\!] \\ \mathcal{T}_C[\![\texttt{pt2}]\!] <:_{\Lambda^P_{<:}} \mathcal{T}_C[\![\texttt{pt1}]\!]\end{array}}{\mathcal{T}_C[\![\texttt{pt1} \to \texttt{rt1}]\!] <:_{\Lambda^P_{<:}} \mathcal{T}_C[\![\texttt{pt2} \to \texttt{rt2}]\!]}$ $\dfrac{\texttt{ty1} = \texttt{ty2}}{\mathcal{T}_C[\![\texttt{ty1}]\!] <:_{\Lambda^P_{<:}} \mathcal{T}_C[\![\texttt{ty1}]\!]}$ Parameter Type Subgoal: $\dfrac{\begin{array}{c}\texttt{rt1} <:_{\mathcal{SOOL}} \texttt{rt2} \\ \texttt{pt2} <:_{\mathcal{SOOL}} \texttt{pt1} \\ \mathcal{T}_C[\![\texttt{rt1}]\!] <:_{\Lambda^P_{<:}} \mathcal{T}_C[\![\texttt{rt2}]\!] \\ \mathcal{T}_C[\![\texttt{pt2}]\!] <:_{\Lambda^P_{<:}} \mathcal{T}_C[\![\texttt{pt1}]\!]\end{array}}{\mathcal{T}_C[\![\texttt{pt2}]\!] <:_{\Lambda^P_{<:}} \mathcal{T}_C[\![\texttt{pt1}]\!]}$ Return Type Subgoal: $\dfrac{\begin{array}{c}\texttt{rt1} <:_{\mathcal{SOOL}} \texttt{rt2} \\ \texttt{pt2} <:_{\mathcal{SOOL}} \texttt{pt1} \\ \mathcal{T}_C[\![\texttt{rt1}]\!] <:_{\Lambda^P_{<:}} \mathcal{T}_C[\![\texttt{rt2}]\!] \\ \mathcal{T}_C[\![\texttt{pt2}]\!] <:_{\Lambda^P_{<:}} \mathcal{T}_C[\![\texttt{pt1}]\!]\end{array}}{\mathcal{T}_C[\![\texttt{rt1}]\!] <:_{\Lambda^P_{<:}} \mathcal{T}_C[\![\texttt{rt2}]\!]}$
auto	Qed.

Figure 2.10: Example: Proving subtypes are preserved.

$\mathcal{SOOL}$	$\Lambda^P_{<:}$ =	$\Lambda^P_{<:}$ $<:$
=	Preserved	Preserved
$<:$	Not Preserved	Preserved

Figure 2.11: Table showing conditions under which Coq should prove subtypes are or are not preserved.

Initial	After transformation and application of $Reflex_{<:}$ rule
$\texttt{ty1} <:_{\mathcal{SOOL}} \texttt{ty2}$	$\texttt{ty1} <:_{\mathcal{SOOL}} \texttt{ty2}$
$\mathcal{T}_C[\![\texttt{ty1}]\!] <:_{\Lambda^P_{<:}} \mathcal{T}_C[\![\texttt{ty2}]\!]$	$\mathcal{T}_C[\![\texttt{ty1}]\!] <:_{\Lambda^P_{<:}} \mathcal{T}_C[\![\texttt{ty2}]\!]$
$\overline{\mathcal{T}_C[\![\texttt{Ref ty1}]\!] <:_{\Lambda^P_{<:}} \mathcal{T}_C[\![\texttt{Ref ty2}]\!]}$	$\overline{Ref\ \mathcal{T}_C[\![\texttt{ty1}]\!] = Ref\ \mathcal{T}_C[\![\texttt{ty2}]\!]}$

Figure 2.12: Third goal produced in impossible proof.

Changing Tactics

We have proved what we wanted to prove, i.e., that subtypes are preserved. But what if we gave a new, incorrect, rule for subtyping of reference types? For example, we could state that $\mathcal{SOOL}$ reference types are subtypes if the types they refer to are subtypes, or, more formally:

$$Ref_{<:} \quad \frac{C \vdash \texttt{S} <: \texttt{T}}{C \vdash \texttt{Ref S} <: \texttt{Ref T}}$$

We wish to show that where the correct rule is retained for $\Lambda^P_{<:}$ the proof must fail, and will do so in an informative fashion. This situation corresponds to the bottom left quadrant of the table in Figure 2.11; the original proof corresponds to the top left quadrant.

We introduce a new Coq rule corresponding to our revised rule for reference types: $\forall \texttt{ty1 ty2}, \texttt{ty1} <:_{\mathcal{SOOL}} \texttt{ty2} \Longrightarrow \texttt{Ref ty1} <:_{\mathcal{SOOL}} \texttt{Ref ty2}$.

The induction step will now yield three subgoals; the first two the same as before and the last goal corresponding to our new $Ref_{<:}$ rule (see Figure 2.12).

We may apply the translation step, just as with function types, replacing the subgoal with the transformed subgoal: $Ref\ \mathcal{T}_C[\![\texttt{ty1}]\!] <:_{\Lambda^P_{<:}} Ref\ \mathcal{T}_C[\![\texttt{ty2}]\!]$. As references are not functions we cannot apply the $Function_{<:}$ rule. The only rule left is $Reflex_{<:}$; we may apply it but this yields an obviously impossible subgoal (see Figure 2.12).

Note that this example is analogous to the problem involving covariant subtyping of arrays discussed in the introduction. Java requires that the types of array elements be subtypes rather than the stricter (and correct) requirement that they be equal. Here we relaxed the requirement for $\mathcal{SOOL}$ reference types in the same way and our proof failed in an informative way. In a fully developed version of Gallimaufry, we could use the same technique on arrays and our proof would fail in the same way.

Now consider what must happen if we move to the right in the table in Figure 2.11, i.e., introduce an incorrect subtyping rule for $\Lambda^P_{<:}$ that corresponds to the incorrect rule for $\mathcal{SOOL}$ that we introduced in the

previous example. We would expect to be able to prove that subtypes are preserved and we can.

Finally, consider the top right quadrant in the table. We would expect to prove that subtypes are preserved in this case since $\mathcal{SOOL}$ requires equality in the types referred to but $\Lambda^{P}_{<:}$ requires only that they are in the subtype relation; we do so using exactly the same tactics that we used to construct our original proof.

Chapter 3

Summary

3.1 Expected Contributions

Gallimaufry is a work in progress. However, we expect that this project will result in the following contributions.

- Most importantly, Gallimaufry is, to our knowledge, the only language extension development framework with an integrated type-safety component and as such is a significant step forward in the area of language design.
- We believe that the technique we are using to develop the proof of correctness of the translation in Gallimaufry is novel. The first step is the development of the translator. The translator is a working program translating programs written in an object-oriented language to semantically equivalent programs written in the lambda calculus. The translator is then itself used as model for definitions suitable for the Coq Proof Assistant [10]. These definitions are the structures used by Coq in the proof. In the first place, the generation direction is unusual [9]; first the translator is built and then the proof is derived from the source code of the translator. This is in the opposite direction to the operation of the Coq program extraction facility. Unlike Krakatoa [21] and Why [34], the proof is extracted directly from the program, rather than specifications of the program's behavior. Recollect that the goal of Gallimaufry is a proof of the type-safety of a language and that the translator translates programs written in that language. The construction of the translator is just a step on the way to the proof. In contrast, the purpose of Krakatoa and Why is to add confidence to an existing program. Having a working translator for a model gives us greater confidence in the Coq definitions that we derive from it. During development of the translator we have detected a few errors in the syntax and typechecking rules of $\mathcal{SOOL}$ as defined in [5]; this emphasizes the importance of the translator development in constructing a correct proof.
- We believe that Gallimaufry is the first tool to automate a proof of type-safety using a *translational* semantics, rather than an operational semantics, a denotational semantics or an axiomatic semantics. Translation is one of the most familiar concepts in programming languages; compilers translate from a higher-level language to a lower-level language and optimizing compilers transform code while

maintaining semantics. We chose a translational semantics for Gallimaufry because we believe that familiarity with the concept of translation will make its use more intuitive for the intended user who may not have a strong background in programming language semantics. By implementing the $\mathcal{SOOL}$ translator, and by automating the proof of type-safety in Coq we hope to demonstrate the soundness of the translational semantics developed in [5]. And by demonstrating a technique for automating such a proof we will have made a significant contribution to the area of type theory.

- The ultimate purpose of Gallimaufry is to allow a language designer to experiment with language features. Gallimaufry will allow the user to introduce language features by describing the syntax and the translation. From these, Gallimaufry will construct the necessary additional Coq definitions and replay the proof using pre-existing tactics and hints as well as some that are suggested by the new language feature. It will then either complete the proof, so that the user is assured of the type safety of the new feature, or indicate that the proof fails and where it runs into trouble. Our tool is aimed at a user who has some familiarity with type theory and the basics of compiler design but who does not necessarily understand the underlying logic of the proof. A significant challenge will be to make this tool useful to such a user. Thus another contribution of our work will be the techniques we develop for extracting meaningful messages from the failure of the proof that will allow the user to correct the syntax or modify the translation of the language feature.

Chapter 4

Plan

4.1 Practical Considerations

4.1.1 Minor Requirements

During the 2005/2006 year a considerable portion of my time and resources will be taken up in completing my minor requirements. I began to fulfill the requirements for a Mathematics minor several years ago; at that time it seemed a reasonable choice for a student who was interested in pursuing a career primarily in teaching. To minor in Mathematics one must choose a specialty field; the one I chose was Algebra. I have completed all the preliminary, i.e., 500 level, courses required; to complete the minor I must now take the advanced, i.e., 700 level, courses. However, now that my research relies so heavily on mathematical logic some formal training in the subject is desirable. Taking the course Math 571, Mathematical Logic, or Math 770, Foundations of Mathematics, would go some way toward acquiring this formal training. Unfortunately, no course in logic will help me to satisfy the requirements for my Mathematics minor in any way. For this reason it is my intention to reclassify my minor as a Distributed one.

4.1.2 Support

I receive support from the university through teaching assistantships. I do not have guaranteed support, and although it is unlikely that I will not receive assistantships for the 2005/2006 school year it is still possible. If I do receive support I will need to perform the duties required of me by my assistantship.

If I do not receive support my situation will be more difficult. Most important is the resulting loss of health care benefits. I have multiple severe spinal injuries which require frequent care. It is therefore necessary that I retain some form of health insurance. Lacking a spouse, I cannot obtain benefits except by paying for them out of pocket. It will be necessary for me to petition for part-time status and to seek out some sort of employment either so that I can get health insurance or so that I will be able to purchase COBRA insurance.

4.2 Schedule

In the following schedule I have attempted to take into account time spent doing classwork and performing the duties of my assistantship.

Extraction Framework

At this point the most urgent task is to build a framework for automating the extraction from O'Caml code to Coq structures. The problem is made more difficult because the extraction must be configurable. I have discussed previously how some O'Caml functions may be translated to Coq Fixpoint definitions; others may not be; still others may be, but may be better translated as Inductive Definitions. These decisions depend not only on the structure of the O'Caml functions, but also on the context in which they are used. The problem is intricate, formal techniques to address it are not readily available, and the O'Caml data structures which represent the abstract syntax tree of an O'Caml program are by design undocumented [7]. Because of the difficulty, I expect to require at least six months to build a useful extractor. Thus, a preliminary working version of the extractor should be completed in the fall of the 2005/2006 school year.

Completing the Proof

The framework described above should enable the following tasks to proceed a good deal more rapidly than otherwise. Currently, a proof of the correctness of the translation exists only in partial form; partial in the sense that the proof is not complete, and also partial in the sense that it is only for a fragment of the $\mathcal{SOOL}$ language. The proof must be completed and expanded to include the entire $\mathcal{SOOL}$ language. I have already had the experience that in adding another part to the proof or another fragment of the $\mathcal{SOOL}$ language I encounter new difficulties in extracting the O'Caml source to Coq structures. At first, the difficulties usually arose when O'Caml source was extracted to Coq structures that Coq did not accept as in the example where the type constraint system's lookup function could not be expressed as a Fixpoint. More recently, however, the Coq structures are accepted by Coq, but during development of the proof it turns out that the structures have been chosen so as to make construction of the proof awkward. The task of completing the proof is likely to alter the requirements for the O'Caml to Coq extractor. Thus this task should proceed more or less concurrently. I expect to resume work on the proof at the start of the Fall 2005 semester and to complete it near the end of the Spring 2006 semester.

The Extension Framework

Recollect that the real purpose of Gallimaufry is to allow the user to experiment with language extensions. Regardless of how the user interface is implemented, designing the underlying engine which must verify properties of the new syntax, reextract Coq structures, rebuild proofs, and finally yield useful feedback in the case that the proof fails is a big task. I hope to combine completing the proof for the entire $\mathcal{SOOL}$ language with developing the extension framework; perhaps when I add the full set of statements to the

SOOL language I will be able to do it using a prototype extension framework. I expect to begin developing the extension framework toward the start of the Spring 2006 semester.

Publications

My work draws on a number of different ideas and techniques and does not fit easily into any single topic area. I presented a paper [23] at CLASE@ETAPS, the Constructive Logic for Automated Software Engineering Workshop, one of the satellite events of ETAPS 2005, the 8th European Joint Conferences on Theory and Practice of Software. However, while this work is clearly an application of constructive logic methods, it could also be interesting to people working in the area of generative programming, type theory, functional programming, and object-oriented languages. The following are some appropriate venues for various parts of my work.

GPCE – Generative Programming and Component Engineering This conference is an appropriate venue for discussion of the techniques of the O'Caml to Coq extractor as well as for discussion of the user interface to Gallimaufry.

ASE – International Conference on Automated Software Engineering

ICFP – International Conference on Functional Programming This conference is an appropriate venue for discussion of the uses of a purely functional subset of O'Caml to act as the source for the O'Caml to Coq extractor.

UITP – Workshop on User Interfaces for Theorem Provers This workshop is an appropriate venue for discussion of user interaction with Gallimaufry, including both definition of language extensions and error reporting.

TLCA – International Conference on Typed Lambda Calculi and Applications

TLDI – Workshop on Types in Language Design and Implementation

FOOL – International Workshops on Foundations of Object-Oriented Languages

ECOOP – European Conference on Object-Oriented Programming

OOPSLA – Object-Oriented Programming, Systems, Languages and Applications

PLDI – Programming Language Design and Implementation

POPL – Symposium on Principles of Programming Languages

ESOP – European Symposium on Programming

I will submit to appropriate conferences as the progress of my research and the submission deadlines allow.

I intend to work on my thesis document concurrently with the development of Gallimaufry. Each step in development of Gallimaufry will require theoretical work to guide the implementation; it is best to commit this work to the thesis document while it is still fresh in my memory.

Finishing Up

Gallimaufry is a novel idea and I expect that it will turn out to be a rich source for research topics in many areas. Some interesting directions to work on are:

1. Developing the user interface to the point that Gallimaufry is useful to its eventual intended user, i.e., someone with some knowledge of type theory but little understanding of proof techniques or the workings of an automated proof assistant.
2. Demonstrating that Gallimaufry can be used to incorporate a non-trivial extension to $\mathcal{SOOL}$. One such candidate is Kim Bruce's notion of *matching*, a relationship that is akin to subtyping, but less restrictive [4, 6].
3. No lambda calculus can express security properties or synchronization properties; however, various type systems have been or are being developed for these properties and various calculi have been developed that abstract these properties. Extending Gallimaufry to such type systems and calculi is another interesting direction for further research.

I intend to focus on the second direction. A working prototype of Gallimaufry and a single significant extension to $\mathcal{SOOL}$ developed and verified using Gallimaufry will constitute the material for an impressive doctoral thesis.

Bibliography

[1] Jim Alves-Foss, editor. *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*. Springer, 1999.

[2] Joshua Bloch. *Effective Java programming language guide*. Sun Microsystems, Inc., 2001.

[3] Egon Börger and Wolfram Schulte. A programmer friendly modular definition of the semantics of Java. In *Formal Syntax and Semantics of Java*, pages 353–404. Springer-Verlag, 1999.

[4] Kim B. Bruce. Increasing Java's expressiveness with ThisType and match-bounded polymorphism. Technical report, Williams College, Williamstown, MA 01267, 1997.

[5] Kim B. Bruce. *Foundations of Object-oriented Languages: Types and Semantics*. MIT Press, 2002.

[6] Kim B. Bruce, Leaf Peteresen, and Adrian Fiech. Subtyping is not a good "match" for object-oriented languages. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97 — The 11th European Conference on Object-Oriented Programming*, volume 1241, pages 104–127, Jyväskylä, Finland, 1997.

[7] Camlp4 - Tutorial. http://caml.inria.fr/pub/docs/tutorial-camlp4/index.html, 2002.

[8] L. Cardelli. Type Systems. In A.B. Tucker, editor, *The Computer Science and Engineering Handbook*, pages 2208–2236. CRC Press, 1997. Chapter 103.

[9] Robert Cartwright and John L. McCarthy. Recursive programs as functions in a first order theory. In *Proceedings of the International Conference on Mathematical Studies of Information Processing*, pages 576–629. Springer-Verlag, 1979.

[10] The Coq Proof Assistant. http://coq.inria.fr/.

[11] Sophia Drossopoulou and Susan Eisenbach. Java is type safe — probably. *Lecture Notes in Computer Science*, 1241, 1997.

[12] Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is the Java type system sound? *Theor. Pract. Object Syst.*, 5(1):3–24, 1999.

[13] Ecma web site. http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[14] Joseph (Yossi) Gil and David H. Lorenz. Design patterns and language design. *Computer*, 31(3):118–120, 1998.

[15] GJ : A Generic Java Language Extension. http://www.cis.unisa.edu.au/˜ pizza/gj/.

[16] Peter Haggar. *Practical Java: Programming Language Guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2000.

[17] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[18] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. A recipe for raw types. In *Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2001.

[19] J2SE 5.0. http://java.sun.com/j2se/1.5.0/.

[20] Rob Johnson and David Wagner. Finding user/kernel pointer bugs with type inference, 2004.

[21] Krakatoa. http://krakatoa.lri.fr/.

[22] Anne Mulhern. Gallimaufry. http://www.cs.wisc.edu/˜ mulhern/gallimaufry.

[23] Anne Mulhern. Gallimaufry: An Automated Framework for Proving Type-Safety. Elsevier, To appear.

[24] Mayur Naik and Jens Palsburg. A type system equivalent to a model checker. In Mooly Sagiv, editor, *Proceedings of the 14th European Symposium on Programming*, volume 3444 of *Lecture Notes in Computer Science*, pages 374–388. Springer-Verlag, 2005.

[25] Tobias Nipkow and David von Oheimb. Javalight is type-safe—definitely. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 161–170. ACM Press, 1998.

[26] Objective Caml. http://www.ocaml.org/.

[27] Benjamin C. Pierce. Types and programming languages. http://www.cis.upenn.edu/˜ bcpierce/tapl/.

[28] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[29] Self home page. http://Research.Sun.COM/self/.

[30] Smalltalk.org. http://www.smalltalk.org.

[31] R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine—Definition, Verification, Validation*. Springer-Verlag, 2001.

[32] Don Syme. Proving Java type soundness. In *Formal Syntax and Semantics of Java*, pages 83–118. Springer-Verlag, 1999.

[33] Peter Thiemann. Towards a type system for analyzing javascript programs. In Mooly Sagiv, editor, *Proceedings of the 14th European Symposium on Programming*, volume 3444 of *Lecture Notes in Computer Science*, pages 408–422. Springer-Verlag, 2005.

[34] Why: A software verification tool. http://why.lri.fr/.

Appendix A

$\mathcal{SOOL}$

The following contains the definition of the $\mathcal{SOOL}$ language as defined in [5].

A.1 $\mathcal{SOOL}$ Types

Type expressions are built from a collection of type constants, $\mathcal{TC}$. $\mathcal{TC}$ contains the type constants `Integer`, `Real`, `Boolean`, `Character`, `String`, `Void`, and `Command`.

$\mathcal{TI}$ is the set of type identifiers.

$\mathcal{L}$ is the set of record labels.

The set, $\mathcal{TYPE}_{\mathcal{SOOL}}(\mathcal{TC}, \mathcal{L}, \mathcal{TI})$, of type expressions of $\mathcal{SOOL}$ over $\mathcal{TC}$, $\mathcal{TI}$, and $\mathcal{L}$ is given by the following context-free grammar.

By convention, $\mathtt{t} \in \mathcal{TI}$, $\mathtt{C} \in \mathcal{TC}$, $\mathtt{l}_i \in \mathcal{L}$ and $\mathtt{T}, \mathtt{T}_i \in \mathcal{TYPE}_{\mathcal{SOOL}}(\mathcal{TC}, \mathcal{L}, \mathcal{TI})$.

$$
\begin{array}{rcl}
\mathtt{T} \in \mathtt{Type} & ::= & \mathtt{C} \mid \\
 & & \mathtt{t} \mid \\
 & & \mathtt{T}_1 \times \ldots \times \mathtt{T}_n \rightarrow \mathtt{T}_{n+1} \mid \\
 & & \mathtt{Ref\ T} \mid \\
 & & \mathtt{ObjectType\ RT} \mid \\
 & & \mathtt{VisObjectType}(\mathtt{RT}_i, \mathtt{RT}_m) \mid \\
 & & \mathtt{ClassType}(\mathtt{RT}_i, \mathtt{RT}_m) \\
\mathtt{RT} \in \mathtt{RType} & ::= & \{\mid \mathtt{l}_1 : \mathtt{T}_1; \ldots; \mathtt{l}_n : \mathtt{T}_n \mid\}
\end{array}
$$

Items generated by `RType` are not legal types of $\mathcal{SOOL}$, but are used to build its object types. Note however that `RType` and `Type` are co-recursive.

A.2 $\mathcal{SOOL}$ Expressions

$\mathcal{SOOL}$ expressions are built from a collection of $\mathcal{SOOL}$ constants, $\mathcal{EC}$. It should be noted that $\mathcal{EC}$ may include higher-order constants, i.e. functions.

$\mathcal{L}$ is the set of labels, used for method and instance variable names.

$\mathcal{EI}$ is the set of expression identifiers.

The set of pre-expressions of $\mathcal{SOOL}$, $\mathcal{PEXP}_{\mathcal{SOOL}}(\mathcal{EC}, \mathcal{L}, \mathcal{EI})$, over $\mathcal{EC}$, $\mathcal{L}$, $\mathcal{EI}$ is given by the following context-free grammar.

By conventions, $\mathtt{id} \in \mathcal{EI}$, $\mathtt{t} \in \mathcal{TI}$, $\mathtt{c} \in \mathcal{EC}$, $\mathtt{l}_i, \mathtt{m}_i \in \mathcal{L}$, and $\mathtt{T}, \mathtt{T}_i \in \mathcal{TYPE}_{\mathcal{SOOL}}(\mathcal{TC}, \mathcal{L}, \mathcal{TI})$.

Prog	::=	Program id; Blk.
Blk ∈ Block	::=	TDs CDs { S return E }
TDs ∈ TDefs	::=	ϵ | type TDL
TDL ∈ TDefLst	::=	t = T | t = T; TDL
CDs ∈ CDefs	::=	ϵ | const CDL
CDL ∈ CDefLst	::=	id: T = E | id: T = E; CDL
E ∈ Exp	::=	id |
		c |
		nil |
		() |
		nop |
		val E |
		ref E |
		$\mathtt{E}(\mathtt{E}_1, \ldots, \mathtt{E}_n)$ |
		$\mathtt{function}(\mathtt{id}_1\colon \mathtt{T}_1, \ldots, \mathtt{id}_n\colon \mathtt{T}_n)\colon \mathtt{T}_{n+1}$ is Blk |
		$\mathtt{class}(\mathtt{R}_i, \mathtt{R}_m)$ |
		new E |
		E ⇐ m |
		E.l |
		$\mathtt{class}$ inherits E modifies $\mathtt{l}_{j_1}, \ldots, \mathtt{l}_{j_m}$ $(\mathtt{R}_i, \mathtt{R}_m)$
R ∈ Rec	::=	$\{\mid \mathtt{l}_1\colon \mathtt{T}_1\ \mathtt{E}_1; \ldots; \mathtt{l}_n\colon \mathtt{T}_n = \mathtt{E}_n \mid\}$
S ∈ Stmt	::=	nop |
		id := E |
		if E then then { $\mathtt{S}_1$ } else { $\mathtt{S}_2$ } |
		while E do { S } |
		$\mathtt{S}_1; \mathtt{S}_2$

Note that Block and Exp are co-recursive, since functions may contain Blks.

A.3 $\mathcal{SOOL}$ Type Constraint System

A type constraint system, C is defined as follows.

1. The empty set, $\emptyset$, is a type constraint system.
2. If C is a type constraint system and $\mathtt{t}$ is a type identifier that does not appear in C or $\mathtt{T}$, then $C \cup \{\ \mathtt{t} = \mathtt{T}\ \}$ is a type constraint system.

The function $C(\mathtt{T})$ is defined as follows.

1. If $\mathtt{t}$ is a type identifier, then $C(\mathtt{t}) = C(\mathtt{T})$ if $\{\ \mathtt{t} = \mathtt{T}\ \} \in C$, otherwise $C(\mathtt{t}) = \mathtt{t}$.
2. If $\mathtt{C}$ is a type constant, then $C(\mathtt{C}) = \mathtt{C}$.
3. $C(\mathtt{T}_1 \times \ldots \times \mathtt{T}_n \rightarrow \mathtt{T}_{n+1}) = C(\mathtt{T}_1) \times \ldots \times C(\mathtt{T}_n) \rightarrow C(\mathtt{T}_{n+1})$.
4. $C(\mathtt{Ref}\ \mathtt{T}) = \mathtt{Ref}\ C(\mathtt{T})$.
5. $C(\mathtt{ObjectType}\ \mathtt{RT}) = \mathtt{ObjectType}\ C(\mathtt{RT})$.
6. $C(\mathtt{VisObjectType}(\mathtt{RT}_i, \mathtt{RT}_m)) = \mathtt{VisObjectType}(C(\mathtt{RT}_i), C(\mathtt{RT}_m))$.
7. $C(\mathtt{ClassType}(\mathtt{RT}_i, \mathtt{RT}_m)) = \mathtt{ClassType}(C(\mathtt{RT}_i), C(\mathtt{RT}_m))$.
8. $C(\{|\ \mathtt{l}_1 : \mathtt{T}_1; \ldots; \mathtt{l}_n : \mathtt{T}_n\ |\}) = \{|\ \mathtt{l}_1 : C(\mathtt{T}_1); \ldots; \mathtt{l}_n : C(\mathtt{T}_n)\ |\}$.

A.4 $SOOL$ Type Rules

A.4.1 Type Rules for Declarations

$$TypeSec \qquad \frac{C,\ \mathcal{E} \vdash \texttt{TDefLst} \diamond C',\ \mathcal{E}'}{C,\ \mathcal{E} \vdash \texttt{type TDefLst} \diamond C',\ \mathcal{E}'}$$

$$TypeDefLst \qquad \frac{C,\ \mathcal{E} \vdash \texttt{t} = \texttt{T} \diamond C_1, \mathcal{E}_1 \quad C_1, \mathcal{E}_1 \vdash \texttt{TDefLst} \diamond C_1, \mathcal{E}_2}{C, \mathcal{E} \vdash \texttt{t} = \texttt{T; TDefLst} \diamond C_2, \mathcal{E}_2}$$

$$TypeDef \qquad C, \mathcal{E} \vdash \texttt{t} = \texttt{T} \diamond C \cup \{\texttt{t} = \texttt{T}\}, C \ \ if \texttt{t} \notin dom(C)$$

$$ConstSec \qquad \frac{C, \mathcal{E} \vdash \texttt{CDefLst} \diamond C', \mathcal{E}'}{C, \mathcal{E} \vdash \texttt{const CDefLst} \diamond C', \mathcal{E}'}$$

$$ConstDefLst \qquad \frac{C, \mathcal{E} \vdash \texttt{id: T} = \texttt{E} \diamond C_1, \mathcal{E}_1 \quad C, \mathcal{E}_1 \vdash \texttt{CDefLst} \diamond C, \mathcal{E}_2}{C, \mathcal{E} \vdash \texttt{id: T} = \texttt{E; CDefLst} \diamond C, \mathcal{E}_2}$$

$$ConstDef \qquad \frac{C, \mathcal{E} \vdash \texttt{E: T} \quad \texttt{id} \notin dom(\mathcal{E})}{C, \mathcal{E} \vdash \texttt{id: T} = \texttt{E} \diamond C, \mathcal{E} \cup \{\texttt{id: T}\}}$$

A.4.2 Type Rules for Expressions

Identifier
$$C, \mathcal{E} \cup \{\mathtt{id}\colon \mathtt{T}\} \vdash \mathtt{id}\colon C(\mathtt{T})$$

Constant
$$C, \mathcal{E} \vdash \mathtt{c}\colon \mathtt{C}$$

Void
$$C, \mathcal{E} \vdash ()\colon \mathtt{Void}$$

Nil
$$C, \mathcal{E} \vdash \mathtt{nil}\colon \mathtt{ObjectType}\ \mathtt{M}$$

Value
$$\frac{C, \mathcal{E} \vdash \mathtt{E}\colon \mathtt{Ref}\ \mathtt{T}}{C, \mathcal{E} \vdash \mathtt{val}\ \mathtt{E}\colon \mathtt{T}}$$

Function
$$\frac{C, \mathcal{E} \cup \{\, \mathtt{id}\colon \mathtt{T}_1, \ldots, \mathtt{T}_n \,\} \vdash \mathtt{Block}\colon \mathtt{T}_{n+1}}{C, \mathcal{E} \vdash \mathtt{function}(\, \mathtt{id}_1\colon \mathtt{T}_1, \ldots, \mathtt{id}_n\colon \mathtt{T}_n\,)\colon \mathtt{T}_{n+1}\ \mathtt{is}\ \mathtt{Block}\colon \mathtt{T}_1 \times \ldots \times \mathtt{T}_n \to \mathtt{T}_{n+1}}$$

Application
$$\frac{C, \mathcal{E} \vdash \mathtt{E}\colon \mathtt{T}_1 \times \ldots \times \mathtt{T}_n \to \mathtt{T}_{n+1},\ C, \mathcal{E} \vdash \mathtt{E}_1\colon \mathtt{T}_1,\ \ldots,\ C, \mathcal{E} \vdash \mathtt{E}_n\colon \mathtt{T}_n}{C, \mathcal{E} \vdash \mathtt{E}(\mathtt{E}_1, \ldots, \mathtt{E}_n)\colon \mathtt{T}_{n+1}}$$

Reference
$$\frac{C, \mathcal{E} \vdash \mathtt{E}\colon \mathtt{T}}{C, \mathcal{E} \vdash \mathtt{ref}\ \mathtt{E}\colon \mathtt{Ref}\ \mathtt{T}}$$

Class
$$\frac{C, \mathcal{E} \vdash \mathtt{inst}\colon \mathtt{IV},\ C, \mathcal{E} \vdash \mathtt{meth}\colon \mathtt{M}}{C, \mathcal{E} \vdash \mathtt{class}(\, \mathtt{inst}, \mathtt{meth}\,)\colon \mathtt{ClassType}(\, \mathtt{IV}, \mathtt{M}\,)}$$

where

- $C' = C \cup \{\, \mathtt{SelfType} = \mathtt{VisObjectType}(\, \mathtt{IV}^{ref}, \mathtt{M}\,)\,\}$
- $\mathcal{E}' = \mathcal{E} \cup \{\, \mathtt{self}\colon \mathtt{SelfType}, \mathtt{close}\colon \mathtt{SelfType} \to \mathtt{ObjectType}\ \mathtt{M}\,\}$
- SelfType does not occur in IV or M

New
$$\frac{C, \mathcal{E} \vdash \mathtt{E}\colon \mathtt{ClassType}\,\{|\, \mathtt{m}\colon \mathtt{T}_1, \ldots, \mathtt{m}_n\colon \mathtt{T}_n \,|\}}{C, \mathcal{E} \vdash \mathtt{new}\ \mathtt{E}\colon \mathtt{ObjectType}\ \mathtt{M}}$$

Message
$$\frac{C, \mathcal{E} \vdash \mathtt{E}\colon \mathtt{ObjectType}\,\{|\, \mathtt{m}\colon \mathtt{T}_1, \ldots, \mathtt{m}_n\colon \mathtt{T}_n \,|\}}{C, \mathcal{E} \vdash \mathtt{E} \Leftarrow \mathtt{m}_i\colon \mathtt{T}_i}$$

Subclass
$$\frac{\begin{array}{c} C, \mathcal{E} \vdash \mathtt{E}\colon \mathtt{ClassType}(\, \mathtt{IV}_{sup}, \mathtt{M}_{sup}\,) \\ C, \mathcal{E} \vdash \mathtt{inst}\colon \mathtt{IV}_{sub},\ C', \mathcal{E}' \vdash \mathtt{meth}\colon \mathtt{M}_s ub \end{array}}{C, \mathcal{E} \vdash \mathtt{class}\ \mathtt{inherits}\ \mathtt{E}\ \mathtt{modifies}\ \mathtt{l}_{i_1}, \ldots, \mathtt{l}_{i_m}\,(\, \mathtt{inst}, \mathtt{meth}\,)\colon \mathtt{ClassType}(\, \mathtt{IV}, \mathtt{M}\,)}$$

where

- $\mathtt{IV} = \mathtt{IV}_{sup} \oplus \mathtt{IV}_{sub}$ and $\mathtt{M} = \mathtt{M}_{sup} \oplus \mathtt{M}_{sub}$
- there is no overlap in the labels occurring in $\mathtt{IV}_{sup}$ and $\mathtt{IV}_{sub}$

- the overlapping labels in $\mathtt{M}_{sup}$ and $\mathtt{M}_{sub}$ are exactly $\mathtt{l}_{i_1}, \ldots, \mathtt{l}_{i_m}$, and the type of each $\mathtt{l}_i$ in $\mathtt{M}_{sub}$ is a subtype of the type of same label in $\mathtt{M}_{sup}$
- $C' = C \cup \{\, \mathtt{SelfType} = \mathtt{VisObjectType}(\, \mathtt{IV}^{ref}, \mathtt{M}\,)\,\}$
- $\mathcal{E}' = \mathcal{E} \cup \{\, \mathtt{self} : \mathtt{SelfType}, \mathtt{close} : \mathtt{SelfType} \to \mathtt{ObjectType}\,\mathtt{M}\,\}$
- SelfType does not occur in IV or M

$$\textit{InstVble} \quad \frac{C, \mathcal{E} \vdash \mathtt{E} : \mathtt{VisObjectType}\,(\mathtt{IVR}, \mathtt{M}\,)}{C, \mathcal{E} \vdash \mathtt{E}.\mathtt{l}_k : \mathtt{T}_k}$$

where $\mathtt{IVR} = \{|\, \mathtt{l}_1 : \mathtt{T}_1, \ldots, \mathtt{l}_n : \mathtt{T}_n \,|\}$ and $1 \le k \le n$

$$\textit{VisObjMessage} \quad \frac{C, \mathcal{E} \vdash \mathtt{E} : \mathtt{VisObjectType}\,(\mathtt{IVR}, \mathtt{M}\,)}{C, \mathcal{E} \vdash \mathtt{E} \Leftarrow \mathtt{m}_j : \mathtt{U}_j}$$

where $\mathtt{M} = \{|\, \mathtt{m}_1 : \mathtt{U}_1, \ldots, \mathtt{m}_n : \mathtt{U}_k \,|\}$ and $1 \le j \le k$

$$\textit{Record} \quad \frac{C, \mathcal{E} \vdash \mathtt{E}_i : \mathtt{T}_i, \text{ for } 1 \le i \le n}{C, \mathcal{E} \vdash \{|\, \mathtt{l}_1 : \mathtt{T}_1 = \mathtt{E}_1, \ldots, \mathtt{l}_n : \mathtt{T}_n = \mathtt{E}_n \,|\} : \{|\, \mathtt{l}_1 : \mathtt{T}_1, \ldots, \mathtt{l}_n : \mathtt{T}_n \,|\}}$$

$$\textit{Block} \quad \frac{C, \mathcal{E} \vdash \mathtt{TDefs} \diamond C_1, \mathcal{E} \quad C_1, \mathcal{E} \vdash \mathtt{CDefs} \diamond C_1, \mathcal{E}_1 \quad C_1, \mathcal{E}_1 \vdash \mathtt{S}\colon \mathtt{Command} \quad C_1, \mathcal{E}_1 \vdash \mathtt{E}\colon \mathtt{T}}{C, \mathcal{E} \vdash \mathtt{TDefs}\ \mathtt{CDefs}\ \{\ \mathtt{S}\ \mathtt{return}\ \mathtt{E};\ \}\colon \mathtt{T}}$$

where T does not contain any type identifiers defined in TDefs

$$\textit{TypeAbbrev} \quad \frac{C, \mathcal{E} \vdash \mathtt{E} : C(\mathtt{T})}{C, \mathcal{E} \vdash \mathtt{E} : \mathtt{T}}$$

A.4.3 Type Rules for Statements

No Op

$$C, \mathcal{E} \vdash \texttt{nop} : \texttt{Command}$$

Assn

$$\frac{C, \mathcal{E} \vdash \texttt{id} : \texttt{Ref T}, \quad C, \mathcal{E} \vdash \texttt{E} : \texttt{T}}{C, \mathcal{E} \vdash \texttt{id} := \texttt{E} : \texttt{Command}}$$

Cond

$$\frac{C, \mathcal{E} \vdash \texttt{E} : \texttt{Boolean}, \quad C, \mathcal{E} \vdash \texttt{S}_1 : \texttt{Command}, \quad C, \mathcal{E} \vdash \texttt{S}_2 : \texttt{Command}}{C, \mathcal{E} \vdash \texttt{if E then \{ S}_1 \texttt{\} else \{ S}_2 \texttt{\} : Command}}$$

While

$$\frac{C, \mathcal{E} \vdash \texttt{E} : \texttt{Boolean}, \quad C, \mathcal{E} \vdash \texttt{S} : \texttt{Command}}{C, \mathcal{E} \vdash \texttt{while E do \{ S \} : Command}}$$

StmtList

$$\frac{C, \mathcal{E} \vdash \texttt{S}_1 : \texttt{Command}, \quad C, \mathcal{E} \vdash \texttt{S}_2 : \texttt{Command}}{C, \mathcal{E} \vdash \texttt{S}_1\texttt{; S}_2 : \texttt{Command}}$$

Program

$$\frac{C, \mathcal{E} \vdash \texttt{Block} : \texttt{Command}}{C, \mathcal{E} \vdash \texttt{Program id; Block} : \texttt{Command}}$$

A.4.4 Subtyping Rules

$$TypeDef_{<:} \quad \frac{\emptyset \vdash C(\mathtt{S}) <: C(\mathtt{T})}{C \vdash \mathtt{S} <: \mathtt{T}}$$

$$Reflex_{<:} \quad C \vdash \mathtt{S} <: \mathtt{S}$$

$$Trans_{<:} \quad \frac{C \vdash \mathtt{S} <: \mathtt{T}, \quad C \vdash \mathtt{T} <: \mathtt{U}}{C \vdash \mathtt{S} <: \mathtt{U}}$$

$$Function_{<:} \quad \frac{C \vdash \mathtt{T}_i <: \mathtt{S}_i, \text{ for } 1 \le i \le n \quad C \vdash \mathtt{S}_{n+1} <: \mathtt{T}_{n+1}}{C \vdash \mathtt{S}_1 \times \ldots \times \mathtt{S}_n \to \mathtt{S}_{n+1} <: \mathtt{T}_1 \times \ldots \times \mathtt{T}_n \to \mathtt{T}_{n+1}}$$

$$Record_{<:} \quad \frac{m \le n \text{ and } C \vdash \mathtt{S} <: \mathtt{T}_i \text{ for all } 1 \le i \le m}{C \vdash \{|\mathtt{l}_i : \mathtt{S}_i, \ldots, \mathtt{l}_n : \mathtt{S}_n|\} <: \{|\mathtt{l}_1 : \mathtt{T}_1, \ldots, \mathtt{l}_m : \mathtt{T}_m|\}}$$

$$Object_{<:} \quad \frac{C \vdash \mathtt{RType}' <: \mathtt{RType}}{C \vdash \mathtt{ObjectType}\ \mathtt{RType}' <: \mathtt{ObjectType}\ \mathtt{RType}}$$

$$Subsumption \quad \frac{C, \mathcal{E} \vdash \mathtt{E} : \mathtt{S}, \quad C \vdash \mathtt{S} <: \mathtt{T}}{C, \mathcal{E} \vdash \mathtt{E} : \mathtt{T}}$$

Appendix B

Polymorphic Lambda Calculus with Subtyping

The following contains the definition of the polymorphic lambda calculus with subtyping as given in [5].

B.1 Kinds

Kind expressions are built from $*$, which represents the kind of all types. More complex kinds are built up according to the following context-free grammar.

$$\kappa \in \mathit{Kind} \quad ::= \quad * \mid \kappa \Rightarrow \kappa'$$

B.2 Types

Type constructor expressions are built from a collection of type constructor constants, $\mathcal{CC}$. $\mathcal{CC}$ contains the type constructor constants *Integer* and *Boolean*.

$\mathcal{CI}$ is the set of constructor identifiers.

$\mathcal{L}$ is the set of record labels.

The type constructor pre-expressions are given by the following context-free grammar.

By convention, $v^\kappa \in \mathcal{CI}$, $c^\kappa \in \mathcal{CC}$, and $l_i \in \mathcal{L}$.

$$
\begin{array}{lll}
\mu, \nu \in \mathit{Constructor} & ::= & v^\kappa \mid \\
& & c^\kappa \mid \\
& & \mathit{Void} \mid \\
& & \mu \to \nu \mid \\
& & \mu_1 \times \ldots \times \mu_n \mid \\
& & \mu_1 + \ldots + \mu_n \mid \\
& & \{\!| \, l_1 : \mu_1; \ldots; l_n : \mu_n \, |\!\} \mid \\
& & \mathit{Ref}\, \mu \mid \\
& & \mathit{Command} \mid \\
& & \lambda v^\kappa.\mu \mid \\
& & \mu\nu \mid \\
& & \forall v^\kappa.T \mid \\
& & \exists v^\kappa.T \mid \\
& & \forall (v^\kappa <: \mu).T \mid \\
& & \exists (v^\kappa <: \mu).T
\end{array}
$$

B.3 Expressions

The expressions are built from a collection of constants, $\mathcal{EC}$. $\mathcal{EC}$ includes the constant $\underline{fix}$ as well as the integers and booleans.

$\mathcal{L}$ is the set of labels used for record names.

$\mathcal{EI}$ is the set of expression identifiers.

The pre-expressions are given by the following context-free grammar.

By convention, $x \in \mathcal{EI}$, $c \in \mathcal{EC}$, $l_i \in \mathcal{L}$.

$$
\begin{array}{rcl}
M \in \mathcal{PLCE} & ::= & x \mid \\
& & c \mid \\
& & <> \mid \\
& & \lambda(x : T).M \mid \\
& & MN \mid \\
& & < M_1, \ldots, M_n > \mid \\
& & proj_i(M) \mid \\
& & case\ M\ of\ x_1 :\ T_1\ then\ E_1 \parallel \ldots \parallel x_n :\ T_n\ then\ E_n \mid \\
& & in_i^{T_1,\ldots,T_n}(M) \mid \\
& & \{\mid l_1 :\ T_1 = M_1, \ldots, l_n :\ T_n = M_n \mid\} \mid \\
& & M.l_i \mid \\
& & ref\ M \mid \\
& & null \mid \\
& & val\ M \mid \\
& & nop \mid \\
& & N := M \mid \\
& & if\ B\ then\ \{M\}\ else\ \{N\} \mid \\
& & M;\ N \mid \\
& & \Lambda\nu^{\kappa}.M \mid \\
& & \Lambda(\nu^{\kappa} <: \mu).M \mid \\
& & M[\mu] \mid \\
& & pack\langle\mu, M\rangle as \exists\nu^{\kappa}.T \mid \\
& & pack\langle\mu, M\rangle as \exists(\nu^{\kappa} <: \nu).T \mid \\
& & open\ M\ as\ \langle\nu^{\kappa}, x\rangle in\ N
\end{array}
$$

B.4 Kind Checking Rules

$$c^{\kappa} :: \kappa$$

$$v^{\kappa} :: \kappa$$

$$Void :: *$$

$$Command :: *$$

$$\frac{\mu :: * \quad \nu :: *}{\mu \to \nu :: *}$$

$$\frac{\mu_1 :: *, \ldots, \mu_n :: *}{\mu_1 \times \ldots \times \mu_n :: *}$$

$$\frac{\mu_1 :: *, \ldots, \mu_n :: *}{\mu_1 + \ldots + \mu_n :: *}$$

$$\frac{\mu_1 :: *, \ldots, \mu_n :: *}{\{|\, l_1 : \mu_1;\ \ldots;\ l_n : \mu_n \,|\}}$$

$$\frac{\mu :: *}{Ref\ \mu :: *}$$

$$\frac{\mu :: \kappa'}{\lambda v^{\kappa} :: \kappa \Rightarrow \kappa'}$$

$$\frac{\mu :: \kappa \Rightarrow \kappa' \quad \nu :: \kappa}{\mu\nu :: \kappa'}$$

$$\frac{\nu :: \kappa}{(\lambda v^{\kappa}.\mu)\nu \cong [\nu/v^{\kappa}]\mu}$$

B.5 Type Checking Rules

B.5.1 Subtyping Rules

$Reflex_{<:}$ $$\frac{\mu :: \kappa}{C \vdash \mu <: \mu}$$

$Transitivity_{<:}$ $$\frac{C \vdash S <: T \quad C \vdash T <: U}{C \vdash S <: U}$$

$Identifier_{<:}$ $$C \cup \{ \nu^{\kappa} <: \mu \} \vdash \nu^{\kappa} <: \mu$$

$Function_{<:}$ $$\frac{C \vdash S <: S' \quad C \vdash T' <: T}{C \vdash S' \to T' <: S \to T}$$

$Product_{<:}$ $$\frac{C \vdash S_i <: T_i, for\ 1 \le i \le n}{C \vdash S_1 \times \ldots \times S_n \times \ldots \times S_{n+m} <: T_1 \times \ldots \times T_n}$$

$Sum_{<:}$ $$\frac{C \vdash S_1 <: T_1, for\ 1 \le i \le n}{C \vdash S_1 + \ldots + S_n <: T_1 + \ldots + T_n + \ldots + T_{n+m}}$$

$Record_{<:}$ $$\frac{C \vdash S_i <: T_i,\ for\ 1 \le i \le n}{C \vdash \{| l_1 : S_1; \ldots; l_n : S_n; \ldots; l_{n+m} : S_{n+m} |\} <: \{| l_1 : T_1; \ldots; l_n : T_n |\}}$$

$Poly_{<:}$ $$\frac{C \vdash S <: T}{C \vdash \forall(\nu^{\kappa}.S <: \forall\nu^{\kappa}.T}$$

$BdPoly_{<:}$ $$\frac{C \cup \{w^{\kappa} <: \mu\} \vdash S <: T}{C \vdash \forall(\nu^{\kappa} <: \mu).S <: \forall(\nu^{\kappa} <: \mu).T}$$

$Exist_{<:}$ $$\frac{C \vdash S <: T}{C \vdash \exists\nu^{\kappa}.S <: \exists\nu^{\kappa}.T}$$

$BdExist_{<:}$ $$\frac{C \cup \{w^{\kappa} <: \mu\} \vdash S <: T}{C \vdash \exists(\nu^{\kappa} <: \mu).S <: \exists(\nu^{\kappa} <: \mu).T}$$

$ConstructorFcns_{<:}$ $$\frac{C \vdash \mu' <: \mu}{C \vdash \lambda\nu^{\kappa}.\mu' <: \lambda\nu^{\kappa}.\mu}$$

$ConstructorApp_{<:}$ $$\frac{\mu :: \kappa \Rightarrow \kappa' \quad \mu :: \kappa' \quad C \vdash \mu' <: \mu}{C \vdash \mu'\mu <: \mu\nu}$$

$Cong_{<:}$ $$\frac{S \cong S' \quad T \cong T' \quad C \vdash S <: T}{C \vdash S' <: T'}$$

B.5.2 Other Rules

Identifier
$$C, \mathcal{E} \cup \{x : T\} \vdash x : T$$

Constant
$$C, \mathcal{E} \vdash c : N$$

Void
$$C, \mathcal{E} \vdash \langle\rangle : Void$$

Function
$$\frac{C, \mathcal{E} \cup \{x : S\} \vdash M : T}{C, \mathcal{E} \vdash \lambda(x : S).M : S \to T}$$

FuncApp
$$\frac{C, \mathcal{E} \vdash M : S \to T \quad C, \mathcal{E} \vdash N : S}{C, \mathcal{E} \vdash MN : T}$$

Product
$$\frac{C, \mathcal{E} \vdash M_i : T_i, \ for\ 1 \le i \le n}{C, \mathcal{E} \vdash \langle M_1, \ldots, M_n \rangle : T_1 \times \ldots \times T_n}$$

Proj
$$\frac{C, \mathcal{E} \vdash M : T_1 \times \ldots \times T_n}{C, \mathcal{E} \vdash proj_i(M) : T_i} for\ 1 \le i \le n$$

Sum
$$\frac{C, \mathcal{E} \vdash M : T_i}{C, \mathcal{E} \vdash in_i^{T_1, \ldots, T_n}(M) : T_1 + \ldots + T_n} for\ 1 \le i \le n$$

Case
$$\frac{C, \mathcal{E} \vdash M : T_1 + \ldots + T_n \quad C, \mathcal{E} \cup \{x_i : T_i\} \vdash E_i : U, \ for\ 1 \le i \le n}{C, \mathcal{E} \vdash case\ M\ of\ x_1 : T_1\ then\ E_1 \parallel \ldots \parallel x_n : T_n\ then\ E_n : U}$$

Record
$$\frac{C, \mathcal{E} \vdash M_i : T_i, \ for\ 1 \le i \le n}{C, \mathcal{E} \vdash \{| l_1 : T_1 = M_1, \ldots, l_n : T_n = M_n |\} : \{| l_1 : T_1; \ldots; l_n : T_n |\}}$$

Selection
$$\frac{C, C \vdash M : \{| l_1 : T_1; \ldots; l_n : T_n |\}}{C, \mathcal{E} \vdash M.l_i : T_i}, for\ 1 \le i \le n$$

Reference
$$\frac{C, \mathcal{E} \vdash M : T}{C, \mathcal{E} \vdash ref\ M : Ref\ T}$$

Null
$$\mathcal{E} \vdash null : Ref\ T$$

Value
$$\frac{C, \mathcal{E} \vdash M : Ref\ T}{C, \mathcal{E} \vdash val\ M : T}$$

Assignment
$$\frac{C, \mathcal{E} \vdash N : Ref\ T \quad C, \mathcal{E} \vdash M : T}{C, \mathcal{E} \vdash N := M : Command}$$

Conditional

$$\frac{C, \mathcal{E} \vdash B : \mathit{Boolean} \quad C, \mathcal{E} \vdash M : T \quad C, \mathcal{E} \vdash N : T}{C, \mathcal{E} \vdash \mathit{if}\ B\ \mathit{then}\ \{\ M\ \}\ \mathit{else}\ \{\ N\ \} : T}$$

Sequencing

$$\frac{C, \mathcal{E} \vdash M : S \quad C, \mathcal{E} \vdash N : T}{C, \mathcal{E} \vdash M;\ N : T}$$

PolyFunc

$$\frac{C, \mathcal{E} \vdash M : T}{C, \mathcal{E} \vdash \Lambda v^{\kappa}.M : \forall v^{\kappa}.T}$$

PolyApp

$$\frac{C, C \vdash M : \forall v^{\kappa}.T \quad \mu :: \kappa}{C, \mathcal{E} \vdash M[\mu] : [\mu/v^{\kappa}]T}$$

Pack

$$\frac{C, \mathcal{E} \vdash M : [\mu/v^{\kappa}]T \quad \mu :: \kappa}{C, \mathcal{E} \vdash \mathit{pack}\ \langle \mu, M \rangle\ \mathit{as}\ \exists v^{\kappa}.T : \exists v^{\kappa}.T}$$

Unpack

$$\frac{C, \mathcal{E} \vdash M : \exists v^{\kappa}.T \quad C, \mathcal{E} \cup \{x : T\} \vdash N : S}{C, \mathcal{E} \vdash \mathit{open}\ M\ \mathit{as}\ \langle v^{\kappa}, x \rangle\ \mathit{in}\ N : S}$$

BdPolyFunc

$$\frac{C \cup \{v^{\kappa} <: \mu\}, \mathcal{E} \vdash M : T \quad \mu :: \kappa}{C, \mathcal{E} \vdash \Lambda(v^{\kappa} <: \mu).M : \forall(v^{\kappa} <: \mu).T}$$

BdPolyApp

$$\frac{C, \mathcal{E} \vdash M : \forall(v^{\kappa} <: \mu).T \quad C \vdash \mu' <: \mu}{C, \mathcal{E} \vdash M[\mu'] : [\mu'/v^{\kappa}]T}$$

BdPack

$$\frac{C, \mathcal{E} \vdash M : [\mu/v^{\kappa}]T \quad C \vdash \mu <: \nu}{C, \mathcal{E} \vdash \mathit{pack}\ \langle \mu, M \rangle\ \mathit{as}\ \exists(v^{\kappa} <: \nu).T : \exists(v^{\kappa} <: \nu).T}$$

BdUnpack

$$\frac{C, \mathcal{E} \vdash M : \exists(v^{\kappa} <: \mu).T \quad C \cup \{v^{\kappa} <: \mu\}, \mathcal{E} \cup \{x : T\} \vdash N : S}{C, \mathcal{E} \vdash \mathit{open}\ M\ \mathit{as}\ \langle v^{\kappa}, x \rangle\ \mathit{in}\ N : S}$$

where v^{κ} does not appear in S

Subsumption

$$\frac{C, \mathcal{E} \vdash M : S \quad C, \mathcal{E} \vdash S <: T}{C, \mathcal{E} \vdash M : T}$$

Congruence

$$\frac{\mathcal{E} \vdash M : T \quad T \cong T'}{\mathcal{E} \vdash M : T'}$$

Appendix C

Translation

C.1 Translating Types

$$
\begin{aligned}
\mathcal{T}_C[\![\mathtt{C}]\!] &\triangleq C \\
\mathcal{T}_C[\![\mathtt{t}]\!] &\triangleq \left\{ \begin{array}{ll} t & if C(\mathtt{t}) = \mathtt{t} \\ \mathcal{T}_C[\![C(\mathtt{t})]\!] & \text{otherwise} \end{array} \right\} \\
\mathcal{T}_C[\![\mathtt{T}_1 \times \ldots \times \mathtt{T}_n \rightarrow \mathtt{T}_{n+1}]\!] &\triangleq \mathcal{T}_C[\![\mathtt{T}_1]\!] \times \ldots \times \mathcal{T}_C[\![\mathtt{T}_n]\!] \rightarrow \mathcal{T}_C[\![\mathtt{T}_{n+1}]\!] \\
\mathcal{T}_C[\![\mathtt{Ref\ T}]\!] &\triangleq Ref\ \mathcal{T}_C[\![\mathtt{T}]\!] \\
\mathcal{T}_C[\![\{|\mathtt{l}_1 : \mathtt{T}_1; \ldots; \mathtt{l}_n : \mathtt{T}_n|\}]\!] &\triangleq \{|\mathtt{l}_1 : \mathcal{T}_C[\![\mathtt{T}_1]\!]; \ldots; \mathtt{l}_n : \mathcal{T}_C[\![\mathtt{T}_n]\!]|\} \\
\mathcal{T}_C[\![\mathtt{VisObjectType(IVR, M)}]\!] &\triangleq \mathcal{T}_C[\![\mathtt{IVR}]\!] \times (\mathcal{T}_C[\![\mathtt{IVR}]\!] \rightarrow \mathcal{T}_C[\![\mathtt{M}]\!]) \\
\mathcal{T}_C[\![\mathtt{ObjectType\ M}]\!] &\triangleq \exists \Upsilon.\Upsilon \times (\Upsilon \rightarrow \mathcal{T}_C[\![\mathtt{M}]\!] \\
\mathcal{T}_C[\![\mathtt{ClassType(IV, M}]\!] &\triangleq \mathcal{T}_C[\![\mathtt{IV}]\!] \times (\mathcal{T}_C[\![\mathtt{VisObjectType(IV}^{ref}\mathtt{, M)}]\!] \rightarrow \mathcal{T}_C[\![\mathtt{M}]\!])
\end{aligned}
$$

C.2 Translating Expressions

The following gives the translation of selected $\mathcal{SOOL}$ expressions.

$\mathcal{T}_C[\![C, \mathcal{E} \vdash \mathtt{id} : \mathtt{T}]\!] \triangleq id$

$\mathcal{T}_C[\![C, \mathcal{E} \vdash \mathtt{c} : \mathtt{T}]\!] \triangleq c$

$\mathcal{T}_C[\![C, \mathcal{E} \vdash () : \mathtt{Void}]\!] \triangleq \langle\rangle$

$\mathcal{T}_C[\![C, \mathcal{E} \vdash \mathtt{val}\ \mathtt{E} : \mathtt{T}]\!] \triangleq val\ \mathcal{T}_C[\![C, \mathcal{E} \vdash \mathtt{E} : \mathtt{Ref}\ \mathtt{T}]\!]$

$\mathcal{T}_C[\![C, \mathcal{E} \vdash \mathtt{ref}\ \mathtt{E} : \mathtt{Ref}\ \mathtt{T}]\!] \triangleq ref\ \mathcal{T}_C[\![C, \mathcal{E} \vdash \mathtt{E} : \mathtt{T}]\!]$

$\mathcal{T}_C[\![C, \mathcal{E} \vdash \mathtt{E}(\mathtt{E}_1, \ldots, \mathtt{E}_n) : \mathtt{T}]\!] \triangleq \mathcal{T}_C[\![C, C \vdash \mathtt{E} : \mathtt{T}_1 \times \ldots \times \mathtt{T}_n \to \mathtt{T}]\!]\ (\langle \mathcal{T}_C[\![C, \mathcal{E} \vdash \mathtt{E}1 : \mathtt{T}_1]\!], \ldots, \mathcal{T}_C[\![C, \mathcal{E} \vdash \mathtt{E}_n : \mathtt{T}_n]\!] \rangle)$

$\mathcal{T}_C[\![C, \mathcal{E} \vdash \mathtt{function}(\mathtt{id}_1 : \mathtt{T}_1, \ldots, \mathtt{id}_n : \mathtt{T}_n) : \mathtt{T}\ \mathtt{is}\ \mathtt{Block} : \mathtt{T}_1 \times \ldots \times \mathtt{T}_n \to \mathtt{T}]\!] \triangleq$
$\lambda(\langle id_1 : \mathcal{T}_C[\![\mathtt{T}_1]\!], \ldots, \mathtt{id}_n : \mathcal{T}_C[\![\mathtt{T}_n]\!] \rangle).\mathcal{T}_C[\![C, \mathcal{E}' \vdash \mathtt{Block} : \mathtt{T}]\!]$
where $\mathcal{E}' = \mathcal{E} \cup \{\mathtt{id}_1 : \mathtt{T}_1, \ldots, \mathtt{id}_n : \mathtt{T}_n\}$

$\mathcal{T}_C[\![C, \mathcal{E} \vdash \{|\mathtt{l}_1 : \mathtt{T}_1 = \mathtt{E}_1; \ldots; \mathtt{l}_n : \mathtt{l}_n : \mathtt{T}_n = \mathtt{E}_n|\} : \{|\ \mathtt{l}_1 : \mathtt{T}_1; \ldots; \mathtt{l}_n : \mathtt{T}_n\ |\}]\!] \triangleq$
$\{|\ \mathtt{l}_1 : \mathcal{T}_C[\![\mathtt{T}_1]\!] = \mathcal{T}_C[\![C, \mathcal{E} \vdash \mathtt{E}_1 : \mathtt{T}_1]\!]; \ldots; l_n : \mathcal{T}_C[\![\mathtt{T}_n]\!] = \mathcal{T}_C[\![C, \mathcal{E} \vdash \mathtt{E}_n : \mathtt{T}_n]\!]|\}$