A Module System for *LOOM*

by

Leaf Eames Petersen

A Thesis
Submitted in partial fulfillment of
of the requirements for the Degree of Bachelor of Arts with Honors
in Computer Science

WILLIAMS COLLEGE
Williamstown, Massachusetts
May 23, 1997

Abstract

A strong module system is a very important language tool for developing software systems. Classes alone do not allow for sufficient levels of abstraction and separate compilation. Modules can be very helpful in organizing code, providing abstraction, and supporting separate compilation. Abstraction makes it difficult to share types between modules, but transparent types can propagate too much information to allow separate compilation. The use of partially abstract types and manifest types can help to avoid these problems.

Earlier work by Robert van Gent and Angela Schuett under the direction of Professor Kim Bruce resulted in the design and implementation of the language **PolyTOIL**, a type safe object-oriented language with strong polymorphic features. $\mathcal{LOOM}$ is a direct descendant of **PolyTOIL** which omits subtyping in favor of a more flexible version of matching, including matching-based subsumption. We give an overview of $\mathcal{LOOM}$ and of a prototype interpreter for the language. Proofs of the complexity of the matching algorithm and the decidability of type checking are presented. We describe the design and implementation of a module system for $\mathcal{LOOM}$, and present an in-depth discussion of the issues that motivated and affected the design process. Formal type checking and semantic rules are given, and the prototype implementation is described. The module system is evaluated, and proposals are made for further work.

Contents

1 Introduction **1**

2 Overview of $\mathcal{LOOM}$ **3**
2.1 Introduction to **PolyTOIL** and matching 3
2.2 Introduction to $\mathcal{LOOM}$ 5
2.3 **MyType** and hash types 7

3 Some theoretical results **10**
3.1 Matching Complexity 11
3.1.1 Equivalence 11
3.1.2 Matching 17
3.2 Decidability of type checking. 21
3.3 Lower bound on complexity 26
3.3.1 The extreme case - example of type size blow up 26
3.3.2 Practical complexity 30

4 Motivations for Modules **31**
4.1 Programming in the Large 31
4.1.1 Name-space management 32
4.1.2 Abstraction control 32
4.1.3 Separate Compilation 35
4.2 Module systems in existing languages 36
4.2.1 **SML** - Transparent Types and Separate Compilation 37
4.2.2 Modula-3 - a strong module system 38

5 Language Design **40**
5.1 Abstraction - Partial, Complete, and Manifest Types 40
5.2 Designing the $\mathcal{LOOM}$ Modules 43
5.2.1 Syntax issues 43
5.2.2 Semantic issues 46

5.3 Modular Type Checking 50
5.3.1 Definitions 51
5.3.2 Modular type assignment rules and axioms 54
5.4 Modular Semantics 57

6 The Interpreter **60**
6.1 Implementation issues 60
6.2 Using the interpreter 61

7 Evaluation of the Language **62**
7.1 Programming in $\mathcal{LOOM}$ 62
7.1.1 Levels of Access 62
7.1.2 Friends 66
7.1.3 Problems with the language 67
7.2 Future Work 68
7.2.1 Multiple interfaces/modules 68
7.2.2 Parameterization over modules 70
7.2.3 Storage allocation on modules? 71

8 Conclusion **72**
8.1 Are modules and classes redundant? 72

A Complete grammar for $\mathcal{LOOM}$ **78**
A.1 Module Syntax 78
A.2 Base Syntax 79

B Complete type checking rules for $\mathcal{LOOM}$ **83**
B.1 Matching rules 84
B.2 Base language type checking rules 84
B.3 Module type checking rules 88
B.4 Algorithmic type checking rules 91

C Complete semantic rules for $\mathcal{LOOM}$ **93**
C.1 Base language semantics 93
C.2 Module semantics 98

D Example $\mathcal{LOOM}$ programs **100**
D.1 Sets with efficient intersection 100
D.2 Type functions vs. hash types - A comparison 102
D.2.1 Points and ColorPoints using TFuncs 102
D.2.2 Points and ColorPoints using hash types 107

List of Figures

2.1 Hash types in $\mathcal{LOOM}$ 6
2.2 Type that can be measured and compared 7
2.3 Object type with contravariant **MyType** 7
2.4 Binary method restriction 8

3.1 Classes that can generate exponentially large types during type checking . . 27
3.2 Type blow up during message send. 28

4.1 Building an efficient set class from an ordered list in **PolyTOIL** 34
4.2 Transparent types in SML 38

5.1 BNF grammar for $\mathcal{LOOM}$ modules 44
5.2 Example $\mathcal{LOOM}$ interface 45
5.3 Old partial revelation syntax 46
5.4 New partial revelation syntax 46
5.5 $\mathcal{M}_B^e, \mathcal{M}_B^i$ after type checking interface B 54
5.6 Example $\mathcal{LOOM}$ module 56

7.1 Modular Sets - Interface 63
7.2 Modular Sets - Implementation 64
7.3 Abstract sets without modules 66

Chapter 1

Introduction

Once upon a time, people who talked to computers spoke in binary. Happily, this is for the most part no longer the case. Even professional programmers can usually ignore the ones and zeros underlying their code. This is very fortunate, as the type of programs being written for computers has changed drastically since the days of flipping toggle switches and watching LED's. Programming problems these days are better described as engineering problems than algorithmic problems. For many years, it was feasible to approach each new program as a distinct entity. By writing each line of code specifically for the particular problem at hand, programmers would eke out every performance advantage they could to facilitate running programs on the relatively inadequate hardware. As time passed however, computers became faster and programs grew larger. It became unwieldy to attempt to design each program entirely from scratch, and software libraries were developed to provide common functionalities. Speed of execution became less critical, and issues like development and maintenance time became important. It became necessary to begin designing programs around common paradigms and structures. The result of this can be seen in the current emergence of software engineering as a discipline.

Engineering has been defined as the construction of large, complex structures such as bridges and buildings through the application of scientific principles. By analogy, software engineering is the construction of large, complex programs, and if programs are the buildings and bridges of software engineering then surely programming languages must be viewed as the bricks from which they are built. It has become increasingly clear over the past decade that many of our "bridges" are being built with very poor "bricks". The challenge of programming language design then is to ensure that the bricks from which our bridges are built are sound.

The approaches to providing strong language support for large scale programming have varied widely. The introduction of object-oriented programming was partially an attempt to deal with this problem, and for the most part, seems to have had a positive effect. Pro-

grammers are becoming more and more accepting of the value of the modular facilities provided by object-oriented languages, and as the use of common libraries of code has become more and more prevalent, the code reuse and organization provided by object systems has become more and more important. It is not clear though that object-oriented facilities alone are sufficient to fully address all of the issues that come up in building large systems. Languages like C++ [ES90] and Eiffel [Mey92] that attempt to support programming-in-the-large through object systems alone are seeming more and more inadequate to the task. C++ in particular provides immensely complicated functionality in an attempt to provide programmers with sufficient flexibility to organize and write their code in a rational way. In the end, neither language really provides sufficient base language support for large scale programming. Indeed, it is not at all clear that such support can be implemented using objects as the only language structure, especially without falling into the C++ trap of ending up with a system of such semantic complexity that few programmers understand all of its behaviors.

Another approach to supporting programming-in-the-large is to use the idea of modules. Languages such as Ada 95,[Joi95] Modula-3 [Har92, Nel91] and **SML** [Mac85, HMM86] have provided very different and interesting module systems, frequently in addition to object-oriented facilities. This approach is interesting because it allows different functionalities to be assigned to different language features, making the semantics easier to understand and less likely to interact in surprising ways. Recent years have seen a good deal of theoretical work being done with modules in an attempt to better understand their nature. This work seems to be paying off with some very promising results.

We were particularly interested in the use of modules within an object-oriented system. The language $\mathcal{LOOM}$, developed during the Summer and Fall of 1995 at Williams College from an earlier language called **PolyTOIL**, is an object-oriented language that has a number of interesting and innovative features. However, it did not seem to provide good support for programming in the large. In an attempt to remedy this, we decided to study the possibility of layering a module system over the base $\mathcal{LOOM}$ language. The design and implementation of this system is the primary focus of this thesis.

Chapter 2 will continue with an introduction to the base $\mathcal{LOOM}$ language and provide some examples of programming in $\mathcal{LOOM}$. In Chapter 3 we present some theoretical results concerning the decidability and complexity of the $\mathcal{LOOM}$ type checking algorithm. Chapter 4 continues with a detailed analysis of the motivations for adding modules to $\mathcal{LOOM}$ and introduces some of the issues that will be dealt with throughout the rest of the thesis. Chapter 5 discusses the central issues involved in the design of the module system and provides a closer look at some of the more important type checking and semantic rules. Chapter 6 talks about some design issues in the actual implementation, and gives instructions for using the $\mathcal{LOOM}$ interpreter. In Chapter 7 we evaluate the successes and failures of the $\mathcal{LOOM}$ module system, and present ideas for future work. Chapter 8 summarizes the results, and draws conclusions from the work.

Chapter 2

Overview of $\mathcal{LOOM}$

The programming language $\mathcal{LOOM}$ is a descendant of an earlier language called **PolyTOIL** [BSvG95]. **PolyTOIL** is a statically typed object-oriented language that was designed to be completely type safe while still providing significantly more flexibility than existing object-oriented languages, such as C++ [ES90], or Object Pascal [Tes85]. It is itself an outgrowth of two other languages – `TOOPLE` [Bru94, BCK94] and `TOIL` [vG93, BvG93]. `TOIL` – **T**yped **O**bject-Oriented **I**mperative **L**anguage – provides the type safety and flexibility of the functional language `TOOPLE` in an imperative language. **PolyTOIL** extends the basic object-oriented functionality of `TOIL` with parametric polymorphism, allowing even greater flexibility in the type system. One part of the parametric polymorphism of **PolyTOIL** is a form of bounded quantification based on an idea called *matching* [Bru93, Bru94, BSvG95].

2.1 Introduction to PolyTOIL and matching

Matching is a relation between types very similar to subtyping, loosely based on the inheritance hierarchy of objects. Recall that the subtyping relation, denoted as $<:$, holds between any two types of which one *subsumes* the other: that is, where elements of the subtype may be treated entirely as being elements of the supertype. The matching relation is a more general relation that essentially parallels the subclassing relation. An object type τ matches an object type σ if an object of type τ could have been generated by a subclass of a class generating objects of type σ. This relation, originally created as part of the subtyping algorithm for `TOIL`, was chosen as the basis of the bounded polymorphic features of **PolyTOIL**and proved to be exceptionally well-suited to use in object-oriented programs. In practice, saying one type matches another is equivalent to saying that any message sent to an object of the first type could also be sent to an object of the second.[1]

[1]Note that this does note hold true for the so called "Hash" types in $\mathcal{LOOM}$ introduced in section 2.2 when **MyType** appears in a contravariant position in the message.

It is not immediately clear what advantage can be found in using matching over subtyping until one considers the interactions of subtyping with another feature of **PolyTOIL** called **MyType**. It can be shown that in order to preserve type safety, an object type τ can only be a subtype of another type σ if (among other things) all changes in the types of the parameters of the methods of σ occur *contravariantly*. In other words, for $\tau <: \sigma$ to be true all parameters to methods in σ must be subtypes of their corresponding parameters in τ. This presents some difficulties when combined with the **PolyTOIL** type system. Within the scope of each object, **PolyTOIL** defines a type identifier **MyType** that represents the type of the object itself. This type is "anchored" to the type of the object in which it appears, so that an appearance of **MyType** in the type of a method inherited from another class represents the type of an instance of the current class, rather than the parent class in which the method was defined. Clearly then, a class which inherits a method with a parameter of type **MyType** will exhibit *covariant* changes in the method's parameter type for any non-trivial inherit, and hence any objects generated from the new class will not be subtypes of objects generated from the parent. In practice, this means that subclasses can no longer be guaranteed to generate subtypes. As a result, more types can be proven to be in the matching relation than the subtyping relation.

Interestingly enough, both matching and **MyType** turn out to be the most central features of **PolyTOIL**. Bounded polymorphism based on matching proves to be a very useful tool in object-oriented programming: so much so that in programming in **PolyTOIL**, we found that it almost completely replaced the use of subtype polymorphism as a mechanism for producing generic code. The **MyType** construct on the other hand is very commonly used to express the type of methods whose type needs to change with the type of the object in which they appear as they are passed down through inheritance. This appears frequently with `copy`-style methods and so-called *binary* methods – methods whose parameter should always be the same type as the receiver. These results suggested to us that subtyping might not be necessary at all. At the same time, we began to feel that presenting programmers with two separate type hierarchies in a single language might be introducing an unwieldy complexity into the language.

One of the most difficult decisions to make in designing a programming language is not so much what features should be included, but rather which features can (should) be excluded. It is not usually clear from the beginning what parts of a language are truly essential, and which parts are simply redundant or worse. Ada has long been noted as a classic case of language overkill - a language which seem to try to support a superset of all possible language features. This does not really seem to be a sound basis for developing a workable language. It seems much more reasonable to try to find a small set of powerful features that can provide all of the functionality needed without unnecessarily cluttering up the language. In looking at **PolyTOIL**, we felt that perhaps the language had strayed into unnecessary complexity. While identification of the subclass/subtype hierarchies in languages such as Eiffel and C++ frequently introduces significant type insecurities, the resulting simplicity of

the subtype relation seems to be popular with programmers. The complexity introduced by separating the inheritance and subtyping hierarchies in **PolyTOIL** was only exacerbated by the presence of language features based upon both relations. It seemed like a worthwhile endeavor to look at the language with an eye for simplification, and given our experience programming in **PolyTOIL** it seemed feasible to forego subtyping altogether.

2.2 Introduction to $\mathcal{LOOM}$

The result of these observations was the development of the language $\mathcal{LOOM}$. $\mathcal{LOOM}$ retains the syntax and bounded polymorphic[2] features of **PolyTOIL** but completely abandons the subtyping relation in favor of an expanded version of matching. As a whole, $\mathcal{LOOM}$ is a much simpler language than its predecessor. In addition to abandoning subtyping, we decided not to allow the types of inherited methods to be changed, which simplifies the language significantly. While this may seem like a step backwards in the movement towards more flexible type systems, we feel that the presence of the **MyType** construct combined with bounded polymorphism provides sufficient flexibility in redefining methods.[3] The ability to change the type of inherited methods is of very limited usefulness given the requirement that domain changes be contravariant, and we have not found a convincing example that cannot be programmed using bounded polymorphism and the **MyType** construct.

The one major functionality that we feared to lose in eliminating subtyping was subtype polymorphism. In object-oriented languages, the ability to have an instance of one type masquerade as its supertype allows for the definition of data structures and operations that treat heterogeneous objects with common functionality homogeneously. This is especially useful for two purposes. The first is for applying functions to subtypes of their intended parameters thereby allowing functions defined on the supertype to be reused on the subtype. The second is for allowing the construction of heterogeneous data structures by defining the type of the elements of the structure to be a common supertype of the various types of the expected elements. Note that while the first problem can be handled fairly elegantly with parametric polymorphism, the second is essentially impossible in **PolyTOIL** without subtype polymorphism.

Subtype polymorphism in object-oriented languages is usually modeled as a form of subsumption. This means that if e has type τ and $\tau <: \sigma$, then e can be said to have type σ. In order to support applications requiring the functionality of subtype polymorphism, $\mathcal{LOOM}$ provides a similar idea using matching. However, $\mathcal{LOOM}$ requires that types intended to behave as such be explicitly tagged with the type constructor #. This is very

[2] Note that unbounded polymorphism has been abandoned in $\mathcal{LOOM}$ (see 5.2.2)

[3] see appendices D.2.1 and D.2.2 for examples of using bounded quantification and **MyType** to redefine method types

```
type    measurable = ObjectType get_val:func():integer; end;
        comparable = ObjectType greater_than:func(#measurable):bool end;
        integer_nums_class = ClassType
                              var me:integer
                              methods VISIBLE
                                 set:proc(integer);
                                 get_val:func():integer;
                                 greater_than:func(#measurable):bool; end;
       INT_NUMS = ObjectType set:proc(integer);
                             get_val:func():integer;
                             greater_than:func(#measurable):bool  end;
const   number_class = class
        var me=0:integer;
        methods  visible
            set=procedure(the_num:integer)
                  begin me:=the_num;
                  end;
            get_val=function():integer
                  begin return(me);
                  end;
            greater_than=function(comp:#measurable):bool
                  begin  return(me>(comp.get_val()));
                  end;
        end:integer_nums_class;
var
    a_num:INT_NUMS;
    b_num:#comparable;
    c_num:#measurable;
    s:BOOLEAN;
begin
  a_num:=new(number_class);
  a_num.set(6);
  b_num:=a_num.clone();
  c_num:=a_num.clone();
  a_num.set(5);
  s:=b_num.greater_than(a_num); --legal, since INT_NUMS <# measurable
--s:=a_num.greater_than(b_num); --illegal, since comparable !<# measurable
  s:=a_num.greater_than(c_num); --legal, since measurable <# measurable
end.
```

Figure 2.1: Hash types in $\mathcal{LOOM}$

similar in functionality to Ada's idea of tagged records [Joi95]. Essentially the idea is that if e has type τ and $\tau <\# \sigma$, then e also has type $\#\sigma$. We refer to types of the form $\#\tau$ as "hash" types, in reference to the constructor.

For example, note that in figure 2.1, `a_num` is an object of type `INT_NUMS`, but since `INT_NUMS` <# `comparable`, `a_num` can be assigned to `b_num`. Either object may be sent the `greater_than` message since both match `comparable` (*i.e.*, both must have at least the `greater_than` method). However, `b_num` may not be sent as a parameter to `greater_than`, since `comparable` does not match `measurable`. In other words, an object of type `#comparable` cannot be guaranteed to have a `get_val` method. On the other hand, `c_num` may not be sent the message `greater_than`, but may be passed as a parameter to it, since it is guaranteed to support `get_val`. In figure 2.2, the type `comp_and_meas` includes the functionality of both `comparable` and `measurable`. As a result, any object to type `#comp_and_meas` could both be sent the message `greater_than` and passed as a parameter to it.

```
type
    comp_and_meas = ObjectType
                        get_val:func():integer;
                        greater_than:func(#measurable):bool
                    end;
```

Figure 2.2: Type that can be measured and compared

2.3 MyType and hash types

This facility provides most of the original functionality of subtype polymorphism. It allows us to write heterogeneous data structures (see appendix D) and to write functions that operate on all objects whose type matches their parameter. However, there is one difficulty with the hash-type idea that has not been revealed in this example. Consider now the object type listed in figure 2.3.

At first glance, this seems much closer to what we want to accomplish with the type `comparable`, or in general any variable v of type `#comparable` – that is, specify that

```
type
    comparable = ObjectType
                     greater_than:func(mytype):bool
                 end;
```

Figure 2.3: Object type with contravariant **MyType**

```
type
    setable = ObjectType set:proc(#mytype) end;

    singleClassType = ClassType  var x:INTEGER;
                               methods visible
                                   getx:func():INTEGER;
                                   set:proc(#mytype);
                                end;

    doubleClassType = ClassType  include singleClassType
                           var y:INTEGER;
                           methods visible
                                gety:func():INTEGER
                           end;
const
    singleClass = class
        var x=0:INTEGER;
        methods  visible
            set=procedure(other:#mytype)
                  begin x:=other.getx();
                  end;
            getx=function():INTEGER begin return x end;
        end:singleClassType;
    doubleClass = class inherit singleClass
        var y:INTEGER;
        methods  visible
            set=procedure(other:#mytype)
                  begin
                     super.set(other);
                     y:=other.gety();
                  end;
            gety=function():INTEGER begin return y end;
        end:doubleClassType;
var
    break:#setable;
begin
    break:=new(doubleClass);
    break.set(new(singleClass));  --run time error!!
end;
```

Figure 2.4: This example shows why binary methods cannot be sent to hash types. Note that the last message send generates a run time error if the type checker allows it to be sent.

we want to have v hold objects of any type which supports the `greater_than` method. Unfortunately, if $\#\sigma$ is the type of an object o then in general we cannot know the type of **MyType** when it appears in σ, since all we know about the actual type of o is that it matches σ. It turns out that for a method m of o (where o is of type $\#\sigma$), we may safely send the message m to o as long as all occurrences of **MyType** in the type of m are positive (that is, appearing in covariant positions). However, if **MyType** appears negatively in the type of m (that is, appears in a contravariant position) we may not safely send the message m to o. To understand the reasons behind this, consider the example code in figure 2.4.

In the example, `singleClass` holds a single integer, whose value can be obtained with a `getx` method. The object can be set to hold the same value as another object whose type matches the current `mytype` using the `set` method. The second class, `doubleClass`, extends the functionality of `singleClass` to include the integer variable `y` and a `gety` method. The `set` method has been updated to get the value of `y` from its parameter. Both of these classes are perfectly legal in *LOOM*.

In the body of the program, `x` is declared to have type `#setable`. Note that both `singleClass` and `doubleClass` generate objects whose type matches `setable`, since both support the `set` method. Therefore it is perfectly legal to assign `x` an instance of `doubleClass`. However, if we allow the binary message `set` to be sent to `x` as shown on the last line of the program, we can run into a problem. The recipient of `set` here is an instantiation of `doubleClass`, which sends the `gety` message to the parameter of its `set` method. Since the parameter is actually an instantiation of `singleClass` that does not support the `gety` method, this message will fail at run time. This is exactly the type of error that static typing is intended to prevent. Clearly then, we cannot allow the `set` message to be sent to objects whose dynamic type is not completely known. In general, we cannot allow any messages with **MyType** parameters (e.g. binary methods) to be sent to hash-typed objects.

This is a fairly significant restriction, since it forces a tradeoff between the ability to assign different values to a variable and the ability to send those values messages. In general, hash types allow for heterogeneous data structures, and permit disparate elements with common functionality to be treated uniformly with respect to that common functionality. However, they also remove more information than one might expect, and consequently force binary methods to be denied to elements whose type is not known precisely. Note though that with subtype polymorphism, we could not even accomplish the first assignment in figure 2.4 since object types with contravariant occurrences of mytype have no non-trivial subtypes. As a result, neither of the classes would generate subtypes of `setable`, and we would be unable do any kind of subsumption here. With hash types, if the type `setable` above included a `getx` method we would be able to send `getx` to `x` safely, or in general send it any message that did not involve a contravariant **MyType**.

Chapter 3

Some theoretical results

In this chapter, we present some complexity and decidability results concerning the algorithms used for type checking the $\mathcal{LOOM}$ language. In particular, we show that type checking is decidable and give a lower bound for the complexity of the procedure. This is particularly important in light of the fact that in general, subtyping for bounded quantification is undecidable. [Pie92]. In section 3.1 we present a proof of the complexity of the equivalence and matching algorithms for $\mathcal{LOOM}$. In section 3.2 we present a proof of the decidability of the type checking algorithm.

In general, we do not provide in depth discussion of the type checking rules and their use in this context. The complete set of rules along with definitions for symbols used in them appear in appendix B. An in depth look at the origin and meaning of the original **PolyTOIL** type checking rules, of which these are a direct descendant, can be found in [BSvG95]. To aid the reader in following the proof, we include here a definition of the collection of types in $\mathcal{LOOM}$ as follows:

Definition 3.0.1 (Types in $\mathcal{LOOM}$) *Let $\mathcal{V}$ be an infinite collection of type variables, $\mathcal{L}$ be an infinite collection of labels, and $\mathcal{C}$ be a collection of type constants which includes at least the type constants Bool and Num. The type expressions with respect to $\mathcal{V}$ and $\mathcal{C}$ are defined as follows:*

1. *If $t \in \mathcal{V} \cup \mathcal{C}$ {COMMAND,UNIT,PROGRAM, $\bot, \top$} then t is a type expression.*
2. *If τ is a type expression, and $\tau \notin$ { COMMAND,UNIT, PROGRAM,$\bot$} then* ref τ *is considered a type expression.*
3. *If σ and τ are type expressions, then so is $func(\sigma)\colon\tau$.*
4. *If $m_1, \ldots, m_n \in \mathcal{L}$ and $\tau_1, \ldots, \tau_n$ are type expressions, then $\{m_1\colon\tau_1; \ldots; m_n\colon\tau_n\}$ is a (record) type expression.*

5. *Let σ, τ_v, and τ_h be record type expressions and $MT \in \mathcal{V}$. Then VisObjType$(\sigma, (\tau_h, \tau_v))$, ClassType VisObjType$(\sigma, (\tau_h, \tau_v))$, and ObjectType τ_v are type expressions. MT is considered to be a bound variable in the last two type expressions.*

6. *If τ is a type expression which corresponds to the type of some object, then $\#\tau$ is a type expression.*

The matching and equivalence algorithms are in fact quite efficient, and can be done in polynomial time. Since the complexity of the type checking algorithm primarily arises from equivalence calls, this would seem to suggest that type checking could also be done fairly efficiently. However, in section 3.3 we demonstrate that the types of expressions may be exponential in their lexical size. As a result, while equivalence and matching are polynomial in the size of the types on which they operate, they are in the worst case exponential in the size of the expressions generating the types. However, expressions that blow up in this fashion are relatively rare and contrived. In general, the type checking algorithm performs very well, as types tend to be linear or at worst polynomial in the size of the expressions.

3.1 Matching Complexity

The matching algorithm for $\mathcal{LOOM}$ follows almost directly from the rules given in section B.1. Note however that these rules rely heavily on the ability to judge two types to be equal. In the case of $\mathcal{LOOM}$, this is handled as structural equivalence, and there is a separate algorithm for judging whether or not two types are equivalent. We notate the relation defined by the structural equivalence rules as $=$, and say that two types σ and τ are equal if and only if $\sigma = \tau$. We denote an algorithmic query of equivalence for two types σ, and τ in the type context $\mathcal{C}$ as $\mathbf{equiv}(\mathcal{C}, \sigma, \tau)$. In most cases, a matching query degenerates almost immediately into one or more equivalence queries, so it should not be surprising that the complexity of the equivalence algorithm is the largest component of the complexity of the matching algorithm.

3.1.1 Equivalence

The proof of the complexity of an equivalence query proceeds by induction on the lexical size of the types being checked. We assume for the sake of simplicity that all type abbreviations are fully expanded and that record fields are stored in a sorted order. As a metric for the size of a type, we will use its lexical size - i.e. the number of tokens needed to print it out in entirety. We define a function $\mathcal{SZ}$ from types to integers such that $\mathcal{SZ}(\sigma)$ denotes the number of lexical tokens needed to encode the type. Note that in general we will count such symbols as *ClassType* as being a single token for counting purposes. This function will be used for the duration of the proof of the complexity of matching as the metric for type size.

Lemma 3.1.1 is necessary for the proof of equivalence. The lemma says firstly that alpha converting a type variable to a type variable does not change the size of the type, and secondly that the alpha-conversion can be done in time linear in the size of the type. We denote a call to the alpha conversion algorithm to replace s with σ in τ as $\mathbf{subst}(\tau, \sigma, s)$, where s is a type identifier and σ is an arbitrary type expression. A full proof of the lemma is omitted, as the lemma follows rather trivially from the algorithm. We do not discuss the alpha conversion algorithm in detail as it is relatively standard, but the essential idea is that the tree representing the type is traversed, and each type identifier is examined for possible replacement. Every time we enter the scope of a new type variable, we check to make sure that the type identifier being replaced has not been scoped out.

Lemma 3.1.1 (Complexity of Alpha Conversion)
Note that for any type τ and type variables r,t the following hold:

- $\mathcal{SZ}(\mathbf{subst}(\tau, r, t)) = \mathcal{SZ}(\tau)$
- $\mathcal{TIME}(\mathbf{subst}(\tau, r, t)) = \mathcal{SZ}(\tau)$

The first follows from the fact that $\mathcal{SZ}(r) = \mathcal{SZ}(t)$ and the definition of $\mathbf{subst}(\tau, r, t)$. *The second can be seen by noting that every recursion in the algorithm removes at least one token from τ, and never performs any expansion.*

Theorem 3.1.2 claims that the complexity of the equivalence algorithm is quadratic in the size of the types on which the query was made. Note that while this is a fairly loose bound under the given assumptions, it may in fact prove to be too tight for a full system involving recursive types, as recursive types do not allow for full expansion. The proof is structured as a case by case induction on the size of the subtyping query.

Theorem 3.1.2 (Equivalence Complexity)
For an equivalence query $\mathbf{equiv}(\mathcal{C}, \sigma, \tau)$, *let:*

1. $s = \mathcal{SZ}(\sigma) + \mathcal{SZ}(\tau)$
2. $k = |\mathcal{C}|$

Then $\mathcal{TIME}(\mathbf{equiv}(\mathcal{C}, \sigma, \tau)) \leq s^2 + s \log k$ for all σ, τ, and hence the equivalence algorithm is $\mathcal{O}(s^2 + s \log k)$.

Note that s is a measure of the size of the query, and that k is a measure of the number of matching constraints in $\mathcal{C}$. For the purposes of a given equivalence query, k remains constant since no equivalence rule adds or subtracts matching constraints from $\mathcal{C}$.

Proof. (By Course of Values Induction on s)

1. Base Cases:

   (a) if $\mathcal{SZ}(\sigma) = \mathcal{SZ}(\tau) = 1$ then both types are single tokens, and equality maybe checked in unit time. So

$$\begin{aligned} \mathcal{TIME}() &= 1 \textit{ and} \\ 2^2 &\leq s^2 + s\log k \qquad (s \geq 2) \textit{ so} \\ \mathcal{TIME}() &\leq s^2 + s\log k \end{aligned}$$

   (b) if $\sigma = \perp$, then $s = (1 + \mathcal{SZ}(\tau))$ and

$$\begin{aligned} \mathcal{TIME}() &= \mathcal{TIME}(\textbf{isObjectType}(\tau)) \\ &\quad \textit{(}\textbf{isObjectType}() \textit{ does at most one lookup in } \mathcal{C} \\ &\qquad \textit{and never recurs)} \\ &\leq \log k + 1 \\ &< s^2 + s\log k \end{aligned}$$

   (c) if $\sigma = \{\}$ and $\tau = \{\}$ then the query is judged true, so

$$\begin{aligned} \mathcal{TIME}() &= 1 \\ &< 16 + 4\log k \qquad (s = 4) \end{aligned}$$

   (d) if ($\mathcal{SZ}(\sigma) = 1$ and $\mathcal{SZ}(\tau) \neq 1$ and $\tau \neq \perp$) or ($\mathcal{SZ}(\tau) = 1$ and $\mathcal{SZ}(\sigma) \neq 1$ and $\sigma \neq \perp$) then the query is false and

$$\begin{aligned} \mathcal{TIME}() &= 1 \\ 2^2 &\leq s^2 + s\log k \qquad (s \geq 2) \\ \mathcal{TIME}() &\leq s^2 + s\log k \end{aligned}$$

2. Ref: $\textbf{equiv}(\mathcal{C}, ref(\sigma), ref(\tau))$
   This essentially strips off a ref constructor from the types and recurs. The proof is mostly algebra.
   let s = $\mathcal{SZ}(ref(\sigma)) + \mathcal{SZ}(ref(\tau))$

$$\begin{aligned} \mathcal{TIME}() &= \mathcal{TIME}(\textbf{equiv}(\mathcal{C}, \sigma, \tau)) + 1 \\ &\leq (\mathcal{SZ}(\sigma) + \mathcal{SZ}(\tau))^2 + (s-2)\log k + 1 \textit{ (by induction)} \\ &\leq (\mathcal{SZ}(ref(\sigma)) + \mathcal{SZ}(ref(\tau)) - 2)^2 + (s-2)\log k + 1 \\ &\leq \mathcal{SZ}(ref(\sigma))^2 + \mathcal{SZ}(ref(\tau))^2 + 2\mathcal{SZ}(ref(\tau))\mathcal{SZ}(ref(\sigma)) \\ &\quad - 4\mathcal{SZ}(ref(\tau)) - 4\mathcal{SZ}(ref(\sigma)) + 4 + (s-2)\log k + 1 \end{aligned}$$

$$
\begin{aligned}
&\leq (\mathcal{SZ}(ref(\sigma)) + \mathcal{SZ}(ref(\tau)))^2 + (s-2)\log k + 5 \\
&\quad - 4\mathcal{SZ}(ref(\tau)) - 4\mathcal{SZ}(ref(\sigma)) \\
&\quad \textit{but } \mathcal{SZ}(ref(\gamma)) \geq 4 \textit{ must be true since} \\
&\quad \mathcal{SZ}(\gamma) \geq 1 \textit{ is always true for all } \gamma. \\
&\leq (\mathcal{SZ}(ref(\sigma)) + \mathcal{SZ}(ref(\tau)))^2 - 16 + (s-2)\log k + 5 \\
&\leq s^2 + (s-2)\log k \\
&\leq s^2 + s\log k
\end{aligned}
$$

3. Bound/Bound: $\mathbf{equiv}(\mathcal{C}, \forall(t <\!\# \gamma).\tau, \forall(s <\!\# \gamma').\tau')$
   Two cases of bounded genericity. The equivalence algorithm needs to check the equivalence of the bounds and of the return types.
   Define

$$
\begin{aligned}
s &= \mathcal{SZ}(\forall(t <\!\# \gamma).\tau) + \mathcal{SZ}(\forall(s <\!\# \gamma').\tau') \\
s' &= \mathcal{SZ}(\gamma) + \mathcal{SZ}(\gamma') \\
s'' &= \mathcal{SZ}(\tau) + \mathcal{SZ}(\tau') \\
\tau_r &= \mathbf{subst}(\tau, r, t) \\
\tau_r' &= \mathbf{subst}(\tau', r, s)
\end{aligned}
$$

   in

$$
\begin{aligned}
\mathcal{TIME}() &= \mathcal{TIME}(\mathbf{equiv}(\mathcal{C}, \gamma, \gamma)) + \mathcal{TIME}(\mathbf{equiv}(\mathcal{C}, \tau_r, \tau_r')) \\
&\quad + \mathcal{TIME}(\mathbf{subst}(\tau, r, t)) + \mathcal{TIME}(\mathbf{subst}(\tau', r, s)) \\
&= ((s')^2 + s'\log k) + ((s'')^2 + s''\log k) + \mathcal{SZ}(\tau) + \mathcal{SZ}(\tau') \\
&\quad \textit{(by induction and lemma 3.1.1)} \\
&= ((s')^2 + s'\log k) + ((s'')^2 + s''\log k) + s'' \textit{ (by def. of s")} \\
&= (s')^2 + (s'')^2 + s'' + (s' + s'')\log k \\
&< (s')^2 + (s'')^2 + 2s's'' + (s' + s'')\log k \\
&< (s' + s'')^2 + (s' + s'')\log k \\
&< s^2 + s\log k \textit{ } (s' + s'' < s. \textit{ Note that } x^2 \textit{ is monotonic})
\end{aligned}
$$

4. Fix/Fix or #/#: $\mathbf{equiv}(\mathcal{C}, \sigma \to \tau, \sigma' \to \tau')$
   Two functions whose formal parameters were declared to be of type σ and σ', where either type may be a hash type. The equivalence algorithm needs to check that the parameter and return types are the same. Note that while the programmer may write

functions that take multiple parameters, all functions are curried internally.
Define

$$\begin{aligned} s &= \mathcal{SZ}(\sigma \to \tau) + \mathcal{SZ}(\sigma' \to \tau') \\ s' &= \mathcal{SZ}(\sigma) + \mathcal{SZ}(\sigma') \\ s'' &= \mathcal{SZ}(\tau) + \mathcal{SZ}(\tau') \end{aligned}$$

in

$$\begin{aligned} \mathcal{TIME}() &= \mathcal{TIME}(\mathbf{equiv}(\mathcal{C}, \sigma, \sigma')) + \mathcal{TIME}(\mathbf{equiv}(\mathcal{C}, \tau, \tau')) \\ &= (s')^2 + s' \log k + (s'')^2 + s'' \log k \ \textit{(by induction)} \\ &< s^2 + s \log k \ (\textit{since} s' + s'' < s) \end{aligned}$$

5. Record: $\mathbf{equiv}(\mathcal{C}, \{m_1{:}\,\sigma_1 \ldots m_n{:}\,\sigma_n\}, \{m_1{:}\,\sigma'_1 \ldots m_k{:}\,\sigma_k\})$
Two records. Since we assume that records are sorted, it only takes unit time to find the corresponding elements.
Define

$$\begin{aligned} s &= \mathcal{SZ}(\{m_1{:}\,\sigma_1 \ldots m_n{:}\,\sigma_n\}) + \mathcal{SZ}(\{m_1{:}\,\sigma'_1 \ldots m_k{:}\,\sigma_k\}) \\ s' &= \mathcal{SZ}(\sigma_1) + \mathcal{SZ}(\sigma'_1) \\ s'' &= \mathcal{SZ}(\{m_2{:}\,\sigma_2 \ldots m_n{:}\,\sigma_n\}) + \mathcal{SZ}(\{m_2{:}\,\sigma'_2 \ldots m_k{:}\,\sigma_k\}) \end{aligned}$$

in

$$\begin{aligned} \mathcal{TIME}() &= \mathcal{TIME}(\mathbf{equiv}(\mathcal{C}, \sigma_1, \sigma'_1)) + \\ &\quad \mathcal{TIME}(\mathbf{equiv}(\mathcal{C}, \{m_2{:}\,\sigma_2 \ldots m_n{:}\,\sigma_n\}, \{m_2{:}\,\sigma'_2 \ldots m_k{:}\,\sigma_k\})) + 1 \\ &= (s')^2 + s' \log k + (s'')^2 + s'' \log k \ + 1 \ \textit{(by induction)} \\ &\leq s^2 + s \log k \\ &\qquad \textit{(as above, by algebra since } s' + s'' < s) \end{aligned}$$

6. VisObjType: $\mathbf{equiv}(\mathcal{C}, VisObj(\sigma, (\tau_1, \tau_2)), VisObj(\sigma', (\tau'_1, \tau'_2)))$
Two internal class representations. The equivalence algorithm must check that each record field is equal.
Define

$$\begin{aligned} s &= \mathcal{SZ}(VisObj(\sigma, (\tau_1, \tau_2))) + \mathcal{SZ}(VisObj(\sigma', (\tau'_1, \tau'_2))) \\ s' &= \mathcal{SZ}(\sigma) + \mathcal{SZ}(\sigma') \\ s'' &= \mathcal{SZ}(\tau_1) + \mathcal{SZ}(\tau'_1) \\ s''' &= \mathcal{SZ}(\tau_2) + \mathcal{SZ}(\tau'_2) \end{aligned}$$

in

$$\begin{aligned}
\mathcal{TIME}() &= \mathcal{TIME}(\mathbf{equiv}(\mathcal{C}, \sigma, \sigma')) \\
&\quad + \mathcal{TIME}(\mathbf{equiv}(\mathcal{C}, \tau_1, \tau_1')) + \mathcal{TIME}(\mathbf{equiv}(\mathcal{C}, \tau_2, \tau_2')) \\
&= (s')^2 + s' \log k + (s'')^2 + s'' \log k + (s''')^2 + s''' \log k \\
&\qquad \textit{(by induction)} \\
&< (s' + s'' + s''')^2 + (s' + s'' + s''') \log k \\
&< s^2 + s \log k \qquad (s' + s'' + s''' < s)
\end{aligned}$$

7. ObjType: $\mathbf{equiv}(\mathcal{C}, \textit{ObjectType } \sigma, \textit{ObjectType } \tau)$
   Two object types. Equivalence algorithm strips off the type constructor and recurs on the embedded records.
   Define

$$\begin{aligned}
s &= \mathcal{SZ}(\textit{ObjectType } \sigma) + \mathcal{SZ}(\textit{ObjectType } \tau) \\
s' &= \mathcal{SZ}(\sigma) + \mathcal{SZ}(\tau)
\end{aligned}$$

   in

$$\begin{aligned}
\mathcal{TIME}() &= \mathcal{TIME}(\mathbf{equiv}(\mathcal{C}, \sigma, \tau)) \\
&= (s')^2 + s' \log k \ \textit{(by induction)} \\
&< s^2 + s \log k \qquad (s' < s)
\end{aligned}$$

8. ClassType: $\mathbf{equiv}(\mathcal{C}, \mathbf{ClassType}\ \sigma, \mathbf{ClassType}\ \tau)$
   Two class types. Equivalence algorithm strips off the constructor and recurs on the internal structure.
   Define

$$\begin{aligned}
s &= \mathcal{SZ}(\mathbf{ClassType}\ \sigma) + \mathcal{SZ}(\mathbf{ClassType}\ \tau) \\
s' &= \mathcal{SZ}(\sigma) + \mathcal{SZ}(\tau)
\end{aligned}$$

   in

$$\begin{aligned}
\mathcal{TIME}() &= \mathcal{TIME}(\mathbf{equiv}(\mathcal{C}, \sigma, \tau)) \\
&= (s')^2 + s' \log k \ \textit{(by induction)} \\
&< s^2 + s \log k \qquad (s' < s)
\end{aligned}$$

9. #: $\mathbf{equiv}(\mathcal{C}, \#\sigma, \#\tau)$
   Two hash types. Strip away the constructor and recur on the types.

Define

$$\begin{aligned} s &= \mathcal{SZ}(\#\sigma) + \mathcal{SZ}(\#\tau) \\ s' &= \mathcal{SZ}(\sigma) + \mathcal{SZ}(\tau) \end{aligned}$$

in

$$\begin{aligned} \mathcal{TIME}() &= \mathcal{TIME}(\mathbf{equiv}(\mathcal{C}, \sigma, \tau)) \\ &= (s')^2 + s' \log k \ \textit{(by induction)} \\ &< s^2 + s \log k \qquad (s' < s) \end{aligned}$$

10. Other:
    Any pair of types which do not fall under one of these rules are not equivalent.

■

3.1.2 Matching

The proof of the complexity of the matching algorithm proceeds by induction on the size of the types in the query. The notation used is essentially identical to that used in the proof of equivalence complexity in section 3.1.1. Note that most of the matching rules degenerate explicitly to a form of record extension. We say that a record type σ *extends* a record type τ (denoted $\mathbf{ext}(\mathcal{C}, \sigma, \tau)$) if σ contains at least everything that is in τ. Extension is also defined for *VisObjType* which is simply a triple of records, each of which must be in the extension relation to their counterpart in order for the triple to be an extensions. Although the matching algorithm handles extension implicitly, the complexity of the **ext**() relation is handled separately for the sake of the proof.

Theorem 3.1.3 (Extension Complexity)
For an extension query of the form $\mathbf{ext}(\mathcal{C}, \sigma, \tau)$*, let* s *and* k *be as defined in theorem 3.1.2. Then* $\mathcal{TIME}(\mathbf{ext}(\mathcal{C}, \sigma, \tau)) \leq s^2 + s \log k$ *and hence the extension algorithm is* $\mathcal{O}(s^2 + s \log k)$*.*

Proof. (By Course of Values Induction on s)
Note that the **ext**() relation is only defined for record types and *VisObjTypes*. We assume for the sake of the proof that records are stored in sorted order. This means that the complexity of finding *all* the corresponding elements between two record is linear in the size of the larger record.

For the duration of this proof, let $s = \mathcal{SZ}(\sigma) + \mathcal{SZ}(\tau)$ for any query of the form $\mathbf{ext}(\mathcal{C}, \sigma, \tau)$.

- Record($<\#$):
  For a query of the form $\mathbf{ext}(\mathcal{C}, \sigma, \tau)$ where σ and τ are record types, we proceed by finding the first element of the τ. If there is no such element, then τ is an empty record and the query is true. If τ does contain a field, say $n_1{:}\,\gamma_1$, then we move forward through σ until we find an $m_i{:}\,\alpha_i$ such that $m_i = n_i$. If there is no such m_i, then the query returns false. Otherwise, we check $\mathbf{equiv}(\mathcal{C}, \gamma_1, \alpha_i)$ and recur with the remainder of the record.

  - Base Case:
    if $\sigma = \{m_1{:}\,\sigma_1 \ldots m_k{:}\,\sigma_k\}$ *and* $\tau = \{\}$ where $k \geq 0$ then

$$\begin{aligned} \mathcal{TIME}(\mathbf{ext}(\mathcal{C}, \sigma, \tau)) &= 1 \qquad (by\ def) \\ s &\geq 4 \qquad (since\ \mathcal{SZ}(\{\}) = 2) \\ \mathcal{TIME}(\mathbf{ext}(\mathcal{C}, \sigma, \tau)) &\leq s^2 + s \log k \end{aligned}$$

  - Inductive Case:
    $\mathcal{TIME}(\mathbf{ext}(\mathcal{C}, \{m_1{:}\,\sigma_1 \ldots m_i{:}\,\sigma_i \ldots m_n{:}\,\sigma_n\}, \{m_i'{:}\,\tau_i \ldots m_k'{:}\,\tau_k\}))$
    define

$$\begin{aligned} s' &= \mathcal{SZ}(\sigma_i) + \mathcal{SZ}(\tau_i) \\ s'' &= \mathcal{SZ}(\{m_{i+1}{:}\,\sigma_{i+1} \ldots m_n{:}\,\sigma_n\}) + \mathcal{SZ}(\{m'_{i+1}{:}\,\tau_{i+1} \ldots m_k'{:}\,\sigma_k\}) \end{aligned}$$

    by definition,
    $\mathcal{TIME}() = \mathcal{TIME}(find(m_i)) + \mathcal{TIME}(\mathbf{equiv}(\mathcal{C}, \sigma_i, \tau_i))$
    $\quad + \mathcal{TIME}(\mathbf{ext}(\mathcal{C}, \{m_{i+1}{:}\,\sigma_{i+1} \ldots m_n{:}\,\sigma_n\}, \{m'_{i+1}{:}\,\tau_{i+1} \ldots m_k{:}\,\tau_k\}))$
    but note that
    1. $\mathcal{TIME}(\mathbf{equiv}(\mathcal{C}, \sigma_i, \tau_i)) \leq (s')^2 + s' \log k$ (by theorem 3.1.2)
    2. $\mathcal{TIME}(\mathbf{ext}(\mathcal{C}, \{m_{i+1}{:}\,\sigma_{i+1} \ldots m_n{:}\,\sigma_n\}, \{m'_{i+1}{:}\,\tau_{i+1} \ldots m_k{:}\,\tau_k\}))$
       $\leq (s'')^2 + s'' \log k$ (by induction)
    3. $\mathcal{TIME}(find(m_i)) = i$ assuming that the m_i is found. Note that if it is not, then the entire query fails in time proportional to the length of the first record, which is certainly $\mathcal{O}(s^2 + s \log k)$. Note also that $s' + s'' + i < s$.

    Therefore ...

$$\begin{aligned} \mathcal{TIME} &\leq (s')^2 + s' \log k + (s'')^2 + s'' \log k + i \\ &< (s' + s'' + i)^2 + (s' + s'' + i) \log k \\ &< s^2 + s \log k \ \textit{(by 3 above)} \end{aligned}$$

- *VisObjType*: This case merely decomposes the *VisObjType* into its record constituents and recurs on each of them.
  $\mathcal{TIME}(\mathbf{ext}(\mathcal{C}, VisObj(\sigma, (\tau_1, \tau_2)), VisObj(\sigma', (\tau_1', \tau_2'))))$
  define

$$\begin{aligned} s' &= \mathcal{SZ}(\sigma) + \mathcal{SZ}(\sigma') \\ s'' &= \mathcal{SZ}(\tau_1) + \mathcal{SZ}(\tau_1') \\ s''' &= \mathcal{SZ}(\tau_2) + \mathcal{SZ}(\tau_2') \end{aligned}$$

  by definition,

$$\begin{aligned} \mathcal{TIME} &= \mathcal{TIME}(\mathbf{ext}(\mathcal{C}, \sigma, \sigma')) + \mathcal{TIME}(\mathbf{ext}(\mathcal{C}, \tau_1, \tau_1')) \\ &\quad + \mathcal{TIME}(\mathbf{ext}(\mathcal{C}, \tau_2, \tau_2')) \\ &\leq (s')^2 + s' \log k + (s'')^2 + s'' \log k + (s''')^2 + s''' \log k \\ &\quad \textit{(by induction)} \\ &\leq (s' + s'' + s''')^2 + (s' + s'' + s''') \log k \\ &< s^2 + s \log k \end{aligned}$$

■

For the proof of the matching complexity, we define a function $\mathbf{TransMatch}(\mathcal{C}, \tau)$ s.t.

$$\forall (t <\# \tau) \in \mathcal{C}, \mathbf{TransMatch}(\mathcal{C}, t) = \begin{cases} \sigma & \textit{if } \tau \textit{ is a type variable and} \\ & \quad \mathbf{TransMatch}(\mathcal{C}, \tau) = \sigma \\ \tau & \textit{otherwise} \end{cases}$$

The use of bounded polymorphism can create chains of type variables in $\mathcal{C}$. The $\mathbf{TransMatch}(\mathcal{C},)$ function finds the upper bound of a chain of type variables. For example if $\mathcal{C} \supseteq \{s <\# t, t <\# \tau\}$ then $\mathbf{TransMatch}(\mathcal{C}, s) = \tau$. Like the $\mathbf{ext}()$ function defined above, $\mathbf{TransMatch}(\mathcal{C}, \tau)$ is handled implicitly in the actual algorithm, but is separated out for the sake of the proof.

Lemma 3.1.4 (Complexity of TransMatch$(\mathcal{C},)$)
The complexity of the $\mathbf{TransMatch}(\mathcal{C},)$ *function is* $\mathcal{O}(d \log k)$ *where* d *is the length of the longest chain in* $\mathcal{C}$*. Note that this follows from the fact that each query to* $\mathcal{C}$ *takes time* $\log k$*, and that the upper bound of the longest chain can be found with* d *queries to* $\mathcal{C}$*.*

We now have all the tools we need for the proof of complexity for the matching algorithm itself. For types σ and τ and given the type constraint system $\mathcal{C}$, we denote an algorithmic query to the matching algorithm as $\mathbf{match}(\mathcal{C}, \tau, \sigma)$. Note that the matching rule for type variables in section B.1 expands a type variable to a full type from $\mathcal{C}$. This means that the complexity of a matching query may depend on the size of a type in $\mathcal{C}$, rather than just on the complexity of the types in the original query. In general, either of the following to conditions are sufficient for the decidability of the matching algorithm to hold:

1. $\forall(t <\!\!\# \tau) \in \mathcal{C}, t$ does not appear in $\mathbf{TransMatch}(\mathcal{C}, \tau)$
2. The matching algorithm never recurs on a substructure of the types in the query.

Note that both rules require that the chain from t to $\mathbf{TransMatch}(\mathcal{C}, \tau)$ not contain any occurrences of t. This is easy to guarantee with correct alpha conversion. Both of these conditions are met by the current matching rules.

Theorem 3.1.5 (Matching Complexity)
For a matching query of the form $\mathbf{match}(\mathcal{C}, \sigma, \tau)$*, let* k *be as defined in theorem 3.1.2, and let* b *and* d *be defined as follows.*

$$\begin{aligned} b &= \max\{\, \mathcal{SZ}(\tau_i) \mid (t_i <\!\!\# \tau_i) \in \mathcal{C} \,\} \\ d &= \textit{length of the longest chain in } \mathcal{C}. \end{aligned}$$

Define $\mathcal{SZ}'()$ *so that*

$$\mathcal{SZ}'(\tau) = \begin{cases} b+1 & \textit{if } \tau \textit{ is a type variable.} \\ \mathcal{SZ}(\tau) & \textit{otherwise} \end{cases}$$

and finally $s = \mathcal{SZ}'(\sigma) + \mathcal{SZ}(\tau)$*. Then* $\mathcal{TIME}(\mathbf{match}(\mathcal{C}, \sigma, \tau)) \leq s^2 + (s+d)\log k$ *and hence the matching algorithm is* $\mathcal{O}(s^2 + (s+d)\log k)$*.*

Proof. (By Cases)

- $\perp$:

$$\begin{aligned} \mathcal{TIME}(\mathbf{match}(\mathcal{C}, \perp, \tau)) &= \mathcal{TIME}(\mathbf{isObjectType}(\tau)) \\ &= \log k \end{aligned}$$

- Type Variable $\setminus \mathit{ObjectType}$: $\mathcal{TIME}(\mathbf{match}(\mathcal{C}, t, \mathit{ObjectType}\ \tau))$
  let

$$\begin{aligned} \mathit{ObjectType}\ \sigma = \quad & \mathbf{TransMatch}(\mathcal{C}, t) \\ s = \quad & \mathcal{SZ}'(t) + \mathcal{SZ}(\mathit{ObjectType}\ \tau) \\ = \quad & b + 1 + \mathcal{SZ}(\mathit{ObjectType}\ \tau) \\ s' = \quad & \mathcal{SZ}(\sigma) + \mathcal{SZ}(\tau) \end{aligned}$$

  Note that $s \geq s' + 2$ will always be true, since $\mathcal{SZ}(\sigma)$ cannot be larger than b.

$$\begin{aligned} \mathcal{TIME}() &= \mathcal{TIME}(\mathbf{TransMatch}(\mathcal{C}, t)) + \mathcal{TIME}(\mathbf{ext}(\mathcal{C}, \sigma, \tau)) \\ &\leq d\log k + (s')^2 + (s')\log k \\ &\qquad \textit{(by lemma 3.1.4 and by theorem 3.1.3)} \\ &\leq (s')^2 + (s'+d)\log k \\ &< (s)^2 + (s+d)\log k \end{aligned}$$

- Type Variable \ Type Variable:
  $\mathcal{TIME}(\mathbf{match}(\mathcal{C}, s, t)) = 1$
  Query is true if $s = t$, false otherwise.

- *ObjectType**ObjectType*: $\mathcal{TIME}(\mathbf{match}(\mathcal{C}, \textit{ObjectType } \sigma, \textit{ObjectType } \tau))$
  let

$$\begin{aligned} s &= \mathcal{SZ}'(\textit{ObjectType } \sigma) + \mathcal{SZ}(\textit{ObjectType } \tau) \\ &= \mathcal{SZ}(\textit{ObjectType } \sigma) + \mathcal{SZ}(\textit{ObjectType } \tau) \textit{ (by definition of } \mathcal{SZ}'()) \\ s' &= \mathcal{SZ}(\sigma) + \mathcal{SZ}(\tau) \end{aligned}$$

$$\begin{aligned} \mathcal{TIME}() &= \mathcal{TIME}(\mathbf{ext}(\mathcal{C}, \sigma, \tau)) \\ &\leq (s')^2 + (s') \log k \textit{ (by theorem 3.1.3)} \\ &< (s)^2 + (s + d) \log k \textit{ (Note } s > s') \end{aligned}$$

■

3.2 Decidability of type checking.

The type checking algorithm closely parallels the type checking rules given in appendix B.2. The significant differences between the formal type checking rules and the algorithmic type checking rules are discussed in appendix B.4. We will denote a call to the type checking algorithm on an expression *exp* in the type context $\mathcal{C}$ with type assignments $\mathbf{E}$ as $\mathbf{typecheck}(\mathcal{C}, \mathbf{E}, exp)$.

Theorem 3.2.1 (Decidability of Type Checking)
For any expression e given a legal type constraint system $\mathcal{C}$ and a legal type assignment $\mathbf{E}$*, a type checking query of the form* $\mathbf{typecheck}(\mathcal{C}, \mathbf{E}, e)$ *terminates.*

Proof. We use the lexical size of the expression being type checked as a metric of the complexity of each call, and show by induction that all recursive calls are made on strictly smaller expressions. We do not explicitly show type equivalence and matching queries for each case, but note that by theorems 3.1.5 and 3.1.2 all such queries must terminate.

- Base cases:

  Calls of the form $\mathbf{typecheck}(\mathcal{C}, \mathbf{E}, e)$ where e is one of `unit`, `cmd`,`integer n`,`real n`,`bool b`,`string s`,`id ‘‘name’’`, do not involve any recursive calls, and therefore terminate.

- Function **typecheck**$(\mathcal{C}, \mathbf{E}, \texttt{function}\ (v{:}\,\sigma)\ \texttt{Block})$

  Let

$$\begin{aligned} k &= \mathcal{SZ}(\text{function}\ (v{:}\,\sigma)\ \mathit{Block}) \\ &= \mathcal{SZ}(\texttt{Block}) + \mathcal{SZ}(\sigma) + 1 \\ k' &= \mathcal{SZ}(\texttt{Block}) \end{aligned}$$

  Assume **typecheck**$(\mathcal{C}, \mathbf{E}, \mathit{exp})$ terminates when $\mathcal{SZ}(\mathit{exp}) < k$. The type checking of a function expression involves a recursive type checking call only on the block of the form **typecheck**$(\mathcal{C}, \mathbf{E} \cup \{v{:}\,\sigma\}, \texttt{Block})$. (Again, while there are several calls to the equivalence and matching algorithms potentially made here, we ignore them for the sake of simplicity, since we have already proved that both algorithms are decidable, and since they are only called a finite number of times.) Therefore, since $k > k'$, **typecheck**$(\mathcal{C}, \mathbf{E} \cup \{v{:}\,\sigma\}, \texttt{Block})$ terminates by induction and hence **typecheck**$(\mathcal{C}, \mathbf{E}, \texttt{function}\ (v{:}\,\sigma)\ \texttt{Block})$ terminates.

- Bounded polymorphic function: **typecheck**$(\mathcal{C}, \mathbf{E}, \texttt{function}\ (t <\!\!\# \gamma)\ \texttt{Block})$

  Let

$$\begin{aligned} k &= \mathcal{SZ}(\text{function}\ (t <\!\!\# \gamma)\ \mathit{Block}) \\ &= \mathcal{SZ}(\texttt{Block}) + \mathcal{SZ}(\gamma) + 3 \\ k' &= \mathcal{SZ}(\texttt{Block}) \end{aligned}$$

  Assume **typecheck**$(\mathcal{C}, \mathbf{E}, \mathit{exp})$ terminates when $\mathcal{SZ}(\mathit{exp}) < k$. Type checking a bounded polymorphic function expression involves a recursive type checking call only on the block of the form **typecheck**$(\mathcal{C} \cup \{t <\!\!\# \gamma\}, \mathbf{E}, \texttt{Block})$. Therefore, since $k > k'$, **typecheck**$(\mathcal{C} \cup \{t <\!\!\# \gamma\}, \mathbf{E}, \texttt{Block})$ terminates by induction and hence **typecheck**$(\mathcal{C}, \mathbf{E}, \texttt{function}\ (t <\!\!\# \gamma{:}\,\sigma)\ \texttt{Block})$ terminates.

- Function application: **typecheck**$(\mathcal{C}, \mathbf{E}, f(M))$

  Let

$$\begin{aligned} k &= \mathcal{SZ}(f(M)) \\ &= \mathcal{SZ}(f) + \mathcal{SZ}(M) + 1 \\ k' &= \mathcal{SZ}(f) \\ k'' &= \mathcal{SZ}(M) \end{aligned}$$

  Assume **typecheck**$(\mathcal{C}, \mathbf{E}, \mathit{exp})$ terminates when $\mathcal{SZ}(\mathit{exp}) < k$. Type checking a function application requires recursive type checking calls on the applied expression and

the actual parameter of the form **typecheck**$(\mathcal{C}, \mathbf{E}, f)$ and **typecheck**$(\mathcal{C}, \mathbf{E}, M)$. But note that since $k' < k$ and $k'' < k$, **typecheck**$(\mathcal{C}, \mathbf{E}, f)$ and **typecheck**$(\mathcal{C}, \mathbf{E}, M)$ terminate by induction. Therefore, **typecheck**$(\mathcal{C}, \mathbf{E}, f(M))$ terminates.

- Bounded polymorphic function application: **typecheck**$(\mathcal{C}, \mathbf{E}, f(\sigma))$

  Let

$$\begin{aligned} k &= \mathcal{SZ}(f(\sigma)) \\ &= \mathcal{SZ}(f) + \mathcal{SZ}(\sigma) + 1 \\ k' &= \mathcal{SZ}(f) \end{aligned}$$

  Assume **typecheck**$(\mathcal{C}, \mathbf{E}, exp)$ terminates when $\mathcal{SZ}(exp) < k$. Type checking a polymorphic function application requires a recursive type checking call on the applied expression of the form **typecheck**$(\mathcal{C}, \mathbf{E}, f)$. Since $k' < k$, this clearly terminates by induction, and hence **typecheck**$(\mathcal{C}, \mathbf{E}, f(\sigma))$ terminates.

- Record: **typecheck**$(\mathcal{C}, \mathbf{E}, \{m_1 = e_1 \colon \tau_1 \ldots m_n = e_n \colon \tau_n\})$

  Let

$$\begin{aligned} k &= \mathcal{SZ}(\{m_1 = e_1 \colon \tau_1 \ldots m_n = e_n \colon \tau_n\}) \\ &= \sum_{i=1}^{n} \mathcal{SZ}(e_i) + \sum_{i=1}^{n} \mathcal{SZ}(\tau_i) + 2n \text{ (\textit{n identifiers and n ``=''s})} \\ k'_i &= \mathcal{SZ}(e_i) \end{aligned}$$

  Assume **typecheck**$(\mathcal{C}, \mathbf{E}, exp)$ terminates when $\mathcal{SZ}(exp) < k$. Note that type checking a record with n fields requires n recursive calls of the form **typecheck**$(\mathcal{C}, \mathbf{E}, e_i)$. Since $\forall i = 1 \ldots n$, $k'_i < k$, each type checking call **typecheck**$(\mathcal{C}, \mathbf{E}, e_i)$ terminates by induction, and hence since n is strictly finite, **typecheck**$(\mathcal{C}, \mathbf{E}, \{m_1 = e_1 \colon \tau_1 \ldots m_n = e_n \colon \tau_n\})$ terminates.

- Projection: **typecheck**$(\mathcal{C}, \mathbf{E}, e.m_i)$

  Let

$$\begin{aligned} k &= \mathcal{SZ}(e.m_i) \\ &= \mathcal{SZ}(e) + 2 \text{ (\textit{1 for ., 1 for m})} \\ k' &= \mathcal{SZ}(e) \end{aligned}$$

  Assume **typecheck**$(\mathcal{C}, \mathbf{E}, exp)$ terminates when $\mathcal{SZ}(exp) < k$. Note that type checking a projection requires a recursive call of the form **typecheck**$(\mathcal{C}, \mathbf{E}, e)$. Since $k' < k$, **typecheck**$(\mathcal{C}, \mathbf{E}, e)$ terminates by induction, and hence **typecheck**$(\mathcal{C}, \mathbf{E}, a.m_i)$ also terminates.

- Class: $\textbf{typecheck}(\mathcal{C}, \mathbf{E}, \texttt{class}(a{:}\,\sigma, (e_h{:}\,\tau_h, e_v{:}\,\tau_v)))$

  Let

$$\begin{aligned} k &= \mathcal{SZ}(\texttt{class}(a{:}\,\sigma, (e_h{:}\,\tau_h, e_v{:}\,\tau_v))) \\ &= \mathcal{SZ}(a) + \mathcal{SZ}(e_h) + \mathcal{SZ}(e_v) + \mathcal{SZ}(\sigma) + \mathcal{SZ}(\tau_h) + \mathcal{SZ}(\tau_v) + 1 \\ k' &= \mathcal{SZ}(a) \\ k'' &= \mathcal{SZ}(e_h) \\ k''' &= \mathcal{SZ}(e_v) \end{aligned}$$

  Assume $\textbf{typecheck}(\mathcal{C}, \mathbf{E}, exp)$ terminates when $\mathcal{SZ}(exp) < k$. Type checking a class requires three recursive calls of the type checking algorithm - one for the instance variables, one for the hidden methods, and one for the visible methods. Since $k' < k$ and $k'' < k$ and $k''' < k$, $\textbf{typecheck}(\mathcal{C}^{IV}, \mathbf{E}, a)$, $\textbf{typecheck}(\mathcal{C}^{Meth}, \mathbf{E}^{Meth}, e_h)$, and $\textbf{typecheck}(\mathcal{C}^{Meth}, \mathbf{E}^{Meth}, e_v)$ terminate by induction, and hence $\textbf{typecheck}(\mathcal{C}, \mathbf{E}, \texttt{class}(a{:}\,\sigma, (e_h{:}\,\tau_h, e_v{:}\,\tau_v)))$ terminates.

  where $\mathcal{C}^{IV} = \mathcal{C} \cup \{\textbf{MyType} <\# \mathit{ObjectType}\ \tau\}$,
  $\mathcal{C}^{METH} = \mathcal{C}^{IV} \cup \{\textbf{SelfType} <\# \textbf{VisObjType}(\sigma, \tau)\}$,
  $E^{METH} = E \cup \{\mathit{self}{:}\,\textbf{SelfType}, \mathit{close}{:}\,\mathit{Func}(\textbf{SelfType}){:}\,\textbf{MyType}\}$

- Inherit: $\textbf{typecheck}(\mathcal{C}, \mathbf{E}, \texttt{class inherit}\ c\ \texttt{modifying}\ v_1, m_1;$
  $(\{v_1 = a_1'{:}\,\sigma_1, v_{m+1} = a_{m+1}{:}\,\sigma_{m+1}\}, \{m_1 = e_1'{:}\,\tau_1, m_{n+1} = e_{n+1}{:}\,\tau_{n+1}\}))$

  Let

$$\begin{aligned} k &= \mathcal{SZ}(\texttt{class inherit}\ c\ \texttt{modifying}\ v_1, m_1; \ldots \\ \ldots &\quad (\{v_1 = a_1'{:}\,\sigma_1, v_{m+1} = a_{m+1}{:}\,\sigma_{m+1}\}, \{m_1 = e_1'{:}\,\tau_1, m_{n+1} = e_{n+1}{:}\,\tau_{n+1}\})) \\ &= \mathcal{SZ}(c) + \mathcal{SZ}(a_1') + \mathcal{SZ}(a_{m+1}) + \mathcal{SZ}(e_1') + \mathcal{SZ}(e_{n+1}) \\ &\qquad + \mathcal{SZ}(\sigma_{m+1}) + \mathcal{SZ}(\tau_1) + \mathcal{SZ}(\tau_{n+1}) + 9 \\ k_c &= \mathcal{SZ}(c) \\ k_{a_1'} &= \mathcal{SZ}(a_1') \\ k_{a_{m+1}} &= \mathcal{SZ}(a_{m+1}) \\ k_{e_1'} &= \mathcal{SZ}(e_1') \\ k_{e_{n+1}} &= \mathcal{SZ}(e_{n+1}) \end{aligned}$$

  Assume $\textbf{typecheck}(\mathcal{C}, \mathbf{E}, exp)$ terminates when $\mathcal{SZ}(exp) < k$. Type checking an inherit requires recursive calls on the additions and modifications.[1] Note now that:

[1] Note that for the sake of simplicity here we assume a single update and a single extend. It is trivial to extend this to multiple updates and extends simply by replacing the individual additions or modifications with records of additions or modifications.

1. Since $k_c < k, k_{a'_1} < k,\ k_{a_{m+1}} < k,\ k_{e'_1} < k$, and $k_{e_{n+1}} < k$,
2. the calls $\mathbf{typecheck}(\mathcal{C}, \mathbf{E}, c)$, $\mathbf{typecheck}(\mathcal{C}^{IV}, \mathbf{E}, a'_1)$, $\mathbf{typecheck}(\mathcal{C}^{IV}, \mathbf{E}, a_{m+1})$, $\mathbf{typecheck}(\mathcal{C}^{Meth}, \mathbf{E}^{Meth}, e'_1)$ and $\mathbf{typecheck}(\mathcal{C}^{Meth}, \mathbf{E}^{Meth}, e'_1)$ all must terminate by induction.
3. Therefore $\mathbf{typecheck}(\mathcal{C}, \mathbf{E}, \texttt{class inherit}\ c\ \texttt{modifying}\ v_1, m_1;$ $(\{v_1 = a'_1 : \sigma_1, v_{m+1} = a_{m+1} : \sigma_{m+1}\}, \{m_1 = e'_1 : \tau_1, m_{n+1} = e_{n+1} : \tau_{n+1}\}))$ terminates.

where $\mathcal{C}^{IV} = \mathcal{C} \cup \{\mathbf{MyType} <\# \mathit{ObjectType}\ \{m_1 : \tau_1; \ldots; m_{n+1} : \tau_{n+1}\}\}$,
$\mathcal{C}^{METH} = \mathcal{C}^{IV} \cup \{\mathbf{SelfType} <\# \mathbf{VisObjType}(\{v_1 : \sigma_1; \ldots; v_{m+1} : \sigma_{m+1}\},$
$\{m_1 : \tau_1; \ldots; m_{n+1} : \tau_{n+1}\})\}$
$E^{METH} = E \cup \{\mathit{self} : \mathbf{SelfType}, \mathit{close} : \mathit{Func}(\mathbf{SelfType}) : \mathbf{MyType},$
$\mathit{super} : \mathbf{SelfType} \rightarrow \{m_1 : \tau_1; \ldots; m_n : \tau_n\}\}$

- New: $\mathbf{typecheck}(\mathcal{C}, \mathbf{E}, \texttt{new}(c))$

  Let

$$\begin{aligned} k &= \mathcal{SZ}(\texttt{new}(c)) \\ &= \mathcal{SZ}(c) + 1 \\ k' &= \mathcal{SZ}(c) \end{aligned}$$

  Assume $\mathbf{typecheck}(\mathcal{C}, \mathbf{E}, \mathit{exp})$ terminates when $\mathcal{SZ}(\mathit{exp}) < k$. Type checking a new of a class requires a recursive type checking call on the class. Since $k' < k$, $\mathbf{typecheck}(\mathcal{C}, \mathbf{E}, c)$ terminates by induction. Therefore, $\mathbf{typecheck}(\mathcal{C}, \mathbf{E}, \texttt{new}(c))$ terminates.

- (#)Message: $\mathbf{typecheck}(\mathcal{C}, \mathbf{E}, o \Leftarrow m)$

  Let

$$\begin{aligned} k &= \mathcal{SZ}(o \Leftarrow m) \\ &= \mathcal{SZ}(o) + 2 \\ k' &= \mathcal{SZ}(o) \end{aligned}$$

  Assume $\mathbf{typecheck}(\mathcal{C}, \mathbf{E}, \mathit{exp})$ terminates when $\mathcal{SZ}(\mathit{exp}) < k$. Type checking a message send requires type checking the recipient of the message. Since $k' < k$, $\mathbf{typecheck}(\mathcal{C}, \mathbf{E}, o \Leftarrow m)$ terminates by induction. Note that we must also do alpha conversion on the return type (recursively unfolding **MyType**) which by lemma 3.1.1 also terminates, and hence $\mathbf{typecheck}(\mathcal{C}, \mathbf{E}, o \Leftarrow m)$ terminates.

■

3.3 Lower bound on complexity

Sections 3.1 and 3.2 show that type checking in $\mathcal{LOOM}$ is decidable. Moreover, they present hope that it is in general possible to do so quite efficiently given the complexity results for the matching and equivalence algorithms. Note though that the matching and equivalence complexities are given in terms of the size of the types on which they are called. In general, the size of types tends to be fairly close to the size of the expression generating them, but in some cases, several factors can interact to cause the type of an expression to blow up exponentially. In particular, cascaded message sends can cause types that never appeared explicitly in the text of the program to be generated and to grow very rapidly with respect to the size of the expression generating them. In this section, we present an example of this and discuss the problem in more detail.

3.3.1 The extreme case - example of type size blow up

Figure 3.1 shows an example of classes that can be used to generate exponentially large types. In the example, `G` and `H` are functions from types to types and `F` is an object type. Note that in $\mathcal{LOOM}$, **MyType** is scoped out at every object boundary. As a result, type functions are the only way to use another object's **MyType** in a nested object type. This ability to alpha convert **MyType** into another object type through renaming is key to this example. Similarly, the classes must be defined in functions which are abstracted over object types and which return the objects generated by the classes to allow objects of type **MyType** to be passed in to other classes. Note that we have simplified things somewhat by not showing the types of the classes. In this example, the class types are only trivially different from the object types since there are no instance variables in the classes. Intuitively, we can think of this example as setting up a sort of chain of classes, with each successive class containing a method `m1` which returns an object generated by the next class in the chain. The last class in the chain (o1) simply returns itself.

This in and of itself is not problematic - note that all of the types are still actually linear in the size of the classes. However, things get more complicated when we begin sending messages to instantiations of the original class and begin generating classes and types on the fly. Figure 3.2 demonstrates the exponential blow up of the type of a cascaded message send to an instance of the class generated in o3. Note that we combine the message and function application type checking rules in the message send expansions for the sake of clarity. In general, the function application rule for functions with unit parameters (i.e. no parameters) simply removes one lexical token from the type.

In the example, `A` is a variable holding an instantiation of the class generated by `o3`. In the first section, we examine what exactly the type of `A` is - that is, what `F` is when it has been expanded out from its abbreviation. Note that this expansion is entirely the evaluation of the type abbreviations and type function applications in `F`. We use the names `Fmytype`

```
type
    G = TFunc[T <# Top;U <# Top] ObjectType m1:func(T,U):mytype end;
    H = TFunc[T <# Top] ObjectType m1:func():G[T,mytype] end;
    F = ObjectType m1:func():H[mytype] end;
const
o1 = function(T <# Top,U <# Top):G[T,U]
    begin
    return new(
    class
        methods VISIBLE
        m1 = function(x:T,y:U):mytype
            begin
            return self;
            end;
        end) --Class
    end; --function

o2 = function(T <# Top):H[T]
    begin
    return new(
    class
        methods VISIBLE
        m1 = function():G[T,mytype]
            begin
            return o1(T,mytype)
            end;
        end) --class
    end; --function

o3 = function():F
    begin
    return new(
    class
        methods VISIBLE
        m1 = function():H[mytype]
            begin
            return o2(mytype);
            end;
        end) --class
    end; --function
```

Figure 3.1: Classes that can generate exponentially large types during type checking

let A be an instantiation of class o3() from figure 3.1

```
A:F
 :ObjectType m1:func():H[mytype] end
 :ObjectType m1:func():ObjectType
                          m1:func():G[Fmytype,mytype]
                       end
  end
  :ObjectType
     m1:func():ObjectType
                 m1:func():ObjectType
                             m1:func(Fmytype,Hmytype):mytype
                           end
               end
   end
We call this expanded version of F by the name BIG

A.m1():(H[mytype])[BIG/mytype]
       :ObjectType m1:func():G[BIG,mytype] end
       :ObjectType
            m1:func():ObjectType
                        m1:func():ObjectType
                                    m1:func(BIG,Hmytype):mytype
                                   end
                      end
        end
We refer to this expansion of H as BIGGER.

A.m1().m1():(G[BIG,mytype])[BIGGER/mytype]
            :ObjectType m1:func():ObjectType
                                     m1:func(BIG,BIGGER):mytype
                                  end
             end
We refer to this last expansion of G as BIGGEST

A.m1().m1().m1()
    :(ObjectType m1:func(BIG,BIGGER):mytype end)[BIGGEST/mytype]
    :ObjectType m1:func(BIG,BIGGER):BIGGEST end
```

Figure 3.2: Type blow up during message send.

and Hmytype to designate the renamings of the mytype parameters to the type functions. So for example, in the type function application H[mytype], mytype refers to the type F. Inside H however, mytype refers to the type H[F] (or in general, H[T] where T is the actual parameter to the type function H). Therefore, before mytype can be substituted in for the formal parameter T, it must be renamed to Fmytype (where Fmytype is a unique type name) to avoid conflicting with the mytype in H. To make the rest of the example easier to read, we refer to the fully expanded type of A as BIG. Note that $\mathcal{SZ}$(BIG) = 28, counting all of the tokens used.[2]

The second part of figure 3.2 shows the full type of the messages send A.m1(). Recall that the method m1 from A returns an instantiation of the class generated in o2, with mytype passed in as the type parameter. Also recall from the type checking rules in appendix B.2 that the inferred type of a message send is the type given for it in the recipient with the inferred type of the recipient (BIG) substituted in for mytype. The result of this substitution and expansion is the complete type of A.m1().[3] We refer to this expanded type as BIGGER. Notice that even though we continue to refer to BIG by name in BIGGER, it actually refers to the full expansion of F. Therefore, $\mathcal{SZ}(BIGGER) = 27 + \mathcal{SZ}(BIG)$, or $\mathcal{SZ}(BIGGER) = 2 * \mathcal{SZ}(BIG) - 1 = 55$.

The third part of figure 3.2 shows the full type of the message send A.m1().m1(). As in the case above, this requires that the type of the recipient of the message be substituted for mytype in the type of the method. Note however that here, the recipient of the final m1() message is the object returned by A.m1(), whose type (from the previous section) is BIGGER. Also note that we have already at this point substituted BIG into the type of A.m1(). We refer to the resulting type as BIGGEST. Note that $\mathcal{SZ}(BIGGEST) = 18 + \mathcal{SZ}(BIGGER) + \mathcal{SZ}(BIG) = 18 + 3{*}\mathcal{SZ}(BIG)$ -1, or $\mathcal{SZ}(BIGGEST) = 101$.

Finally, consider the last part of figure 3.2. This represents the full type of the message send A.m1().m1().m1(). Once again we must substitute in the type of the object for mytype. Note that from the previous paragraph we know that the type of A.m1().m1() is BIGGEST. Therefore, we substitute BIGGEST in for mytype and get back the type ObjectType m1:func(BIG,BIGGER):BIGGEST end. This is the size of the return type of the function call as listed. Note that:

$$\begin{aligned}
&\mathcal{SZ}(\texttt{ObjectType m1:func(BIG,BIGGER):BIGGEST end}) = \\
&\qquad 9 + \mathcal{SZ}(BIGGEST) + \mathcal{SZ}(BIGGER) + \mathcal{SZ}(BIG) \\
&= \quad 9 + (17 + 3 * \mathcal{SZ}(BIG)) + 2 * \mathcal{SZ}(BIG) - 1 + \mathcal{SZ}(BIG) \\
&= \quad 25 + 6 * \mathcal{SZ}(BIG)
\end{aligned}$$

[2] In general, it is fairly arbitrary which tokens we count, so long as we are consistent and count at least each token that is essential to the representation. For the sake of simplicity here, we will count every token used even though it is not clear that all of these tokens are essential to the representation.

[3] Again, note that we are implicitly applying the function application rule here, since technically, the type of A.m is a function type which must then be applied to the unit parameter

$$= \quad 193$$

Empirically, this is very large. More generally, note that the size of the type is approximately doubling with every message send. In general, the size of a cascaded message send expression can be described as roughly kn, where k is a constant and n is the number of cascaded messages. Given this view of message sends in an example like this, the size of the type grows roughly as 2^n.

To look at the problem more abstractly, note that substitution of a full type for `mytype` can in fact *square* the size of the type being substituted into, since the type may be contain an arbitrary number of occurrences of `mytype` (specifically, enough so that the size of the type is approximately equal to the number of occurrences of `mytype`). Moreover, note that the application of the method does not necessarily remove significant complexity from the type, since the bulk of the size may always be made to appear in the return type of the method (up to an arbitrary point). In general then, for a cascaded message send, we may in principle square the size of the type after each send, resulting in a final type of size $\mathcal{O}(2^{2^n})$.

Note that the example given relies particularly on the ability to abstract over types to propagate each `mytype` to the innermost level of the type. It is not clear whether or not this is an essential property of examples of exponential type growth, but it seems likely. In general, without the ability to propagate specific `mytype` variables through multiple scopes we cannot propagate expansions through multiple message sends. Note that every message send must remove all free occurrences of `mytype` from its type via expansion. Without the ability to explicitly bind `mytype` in subsequent scopes, all the complexity introduced by the expansion must appear only in the current scope, which is removed by the next message send.

3.3.2 Practical complexity

This a very discouraging result. While type checking is not exceptionally time critical, exponential complexity is very problematic even on short examples. If this complexity were to reflect the average complexity of type checking in $\mathcal{LOOM}$, the language would clearly be impractical for general use. In practice however, examples like this are not at all common - in fact they are quite difficult to contrive. Generally speaking, the size of types tends to be linear in the size of the expressions generating them, and type checking proceeds fairly efficiently. Note too that this problem is not unique to $\mathcal{LOOM}$. Type checking in **SML** is also in principle potentially exponential, but is in practice quite manageable. [KM89]

Chapter 4

Motivations for Modules

4.1 Programming in the Large

One thing that we have avoided dealing with up until this point is the issue of programming in the large. There are a number of issues that come up in the course of attempting to build very large systems that are not necessarily addressed by standard programming language constructs. In essence, it is no longer just programming that we need to support, but rather program development on a grand scale. Many different languages, particularly in the object oriented paradigm, have attempted to address this issue, some with explicit extensions to the language and some without. Languages such as C++, Smalltalk, and Eiffel do not have any extra module systems, choosing instead to use classes as the fundamental unit. Other languages, such as Modula-3 are fundamentally structured around the idea of a module. Standard ML provides a small typed language on top of the core language explicitly for handling modular structures. All of these mechanisms are attempts to provide support in some way for programming in the large. Jones [Jon96] identifies three major issues that a module system needs to address:

1. The need to provide a way of organizing code into distinct units in a coherent way to facilitate code and namespace management.

2. Providing abstraction barriers to lessen the dependence of units on the implementation details of other units and provide language support for control of information propagation.

3. Providing support for separate compilation, thereby making the development process easier and making it possible to provide reusable code in libraries without granting access to the source code.

As these issues were central to our design of the $\mathcal{LOOM}$ module system, it is useful to look at each closely, in the interests of both understanding the issues better, and of understanding

why they are not addressed with existing language features.

4.1.1 Name-space management

Facilities for managing name spaces are one of the most commonly supported modular features in current programming languages. It is very important when developing large systems to have some means of organizing code logically and for controlling the name-space. If the scope of all or most of the names used in a program is the program itself, then it becomes quite easy to generate interference between names. This can be particularly troublesome when trying to combine code written by more than one person into a coherent whole. While a consistent naming scheme can alleviate this somewhat, it is difficult to maintain a naming scheme when using code from many different sources without some explicit language support. Classes provide a fairly reasonable form of name-space management for their own operations. Methods do not interfere since they are always qualified by the name of an instantiation of the class to which they apply (although there are some difficult interference questions that come up in languages with multiple inheritance). However, the space of classes, constants, functions, and types remains global in **PolyTOIL**. Moreover, there is no real way of organizing the code into logically coherent units - all **PolyTOIL** programs are essentially one large unit. This problem is fairly common in object-oriented programming languages.

Name-space management is an especially crucial issue when it comes to types in **PolyTOIL**. Separate types must be provided for both a class and the objects it generates. Despite the use of a Rapide [KLM94] style type inclusion mechanism, type sections in **PolyTOIL** programs tend to be large and unwieldy even for small problems. The name-space difficulties this causes are not at all insignificant. This seems perhaps unnecessarily confusing, and is certainly not appropriate for programming in the large - the chances of generating name conflicts when combining code is significant. Moreover, one common criticism of structural typing is that there exists a possibility of unknowingly encountering an accidental type equivalence. While it is not clear that this is as common a problem as it is sometimes made out to be, it is much more likely to occur in a global environment than in a modular environment.

4.1.2 Abstraction control

The second issue - providing useful abstraction barriers - is at the same time one of the most difficult and most interesting of the issues we examined. Relatively few current languages do this well. For the most part, it has proved very difficult to design an abstraction mechanism that supports both implementation-independent modules and efficient coding, two things that are sometimes very much in opposition. The idea is to provide a language facility for hiding information about the implementation of a unit from clients of the unit. This allows

changes in the implementation to take place without affecting the rest of the program. To a certain extent, classes in **PolyTOIL** can provide this in that the types of the objects they generate do not contain the instance variables and hidden methods of the object. This means that so long as two classes provide the same public methods, objects generated by them are interchangeable (i.e. have the same type), and hence have abstracted away implementation details. This is *not* true of most object-oriented languages such as C++ and Eiffel, which allow instance variables to appear in the interfaces and tie subtyping to inheritance. This approach is in many ways fundamentally limited. It is not clear that two different implementations can be handled equally well with the same interface, especially given the fact that even other instantiations of the same class must still only utilize the public interface.

For example, in figure 4.1 a set class may wish to provide an efficient destructive intersection method by taking advantage of its knowledge of the internal representation. The receiver of the `intersect` message holds the intersection of the original value of the receiver and the parameter. In order to do this though, it must grant access to its internal representation to all users through methods like `deleteCur`, `find`, etc. that are necessary for the efficient implementation of intersection. This means that the object can no longer guarantee invariants about itself since all users may now get at the representation. Further, if the internal representation is changed to bit fields, for example, we must either change the public interface as well or else suffer the gross inefficiency of having to convert the information to a different implementation every time we use the old set interface.

The problem is really that there is no general idea of type abstraction. **PolyTOIL** is based around the idea of structural typing and hence has very limited facilities for using named types, despite their growing importance in the language. In the original TOIL implementation, type names were treated entirely as abbreviations for the corresponding full structural types, with no semantic distinction made between a type name and the type itself. All type names could be macro-expanded out during type-checking. With the advent of **PolyTOIL** however, things became more complicated. **PolyTOIL** introduced limited forms of type abstraction in the form of bounded and universal quantification over types. This means that for certain type names, (specifically type variables), little or no type information is available during type checking: specifically, in the case of bounded quantification the system has an upper bound on the type of the parameter, while in the case of universal quantification the system may only assume that the parameter is in fact a type. As a result, all type judgments on these types need to be based almost exclusively on the type name. Making things even more complicated, subsequent modifications to the **PolyTOIL** interpreter included the addition of recursive types. With recursive types, macro-expansion becomes completely infeasible, especially since the nature of the recursive subtyping algorithm is such that multiple expansions of a type may be necessary before a judgment can be made.

In short, it is quite clear that named types are already an integral part of the language.

```
OrdListType = ObjectType
   first: proc(); --move to the first element (off if empty)
   next: proc(); --move to the next elt (off if at end or off)
   off: func():Boolean; -- is current elt off end of list?
   add: proc(Integer); --add an elt, maintaining ordering
   deleteCur: proc();  -- current is next elt after deleteCur
   contains: func(Integer):Boolean; --is param in list (bsearch)
   getCur: func():Integer --get the current element
end;

IntSetType = ObjectType include OrdListType
   remove :proc(Integer);  --remove an element
   intersect: proc(MyType) --receiver contains the intersection
end;                       --note intersect is destructive!

ListSetClass = class inherit OrdListClass
   methods visible
      procedure remove(elt:Integer) is
         begin
            if find(elt) then deleteCur()
         end;
      procedure intersect(other:MyType) is
         begin
            first();
            other.first();
            while (not off()) and (not other.off()) do
               if getCur() < other.getCur() then
                  deleteCur()
               elsif getCur() > other.getCur then
                  other.next()
               else
                  next();
                  other.next()
               end
            end -- while
            while not off() do
               deleteCur()
            end -- while
         end -- function
    end;  -- class
```

Figure 4.1: Building an efficient set class from an ordered list in **PolyTOIL**

Unfortunately, **PolyTOIL** does not provide especially powerful facilities for defining named types. The only place in which named types in **PolyTOIL** can be defined is in a global type section, from which all information is exported indiscriminately. There are no mechanisms for scoping type names, and more importantly, the structural information associated with type names. This is one of the central issues that we felt we needed to address with $\mathcal{LOOM}$.

4.1.3 Separate Compilation

The last issue we want to address is that of separate compilation. The issue of separate compilation is very important to software developers for a number of reasons. Firstly, the time needed to completely compile a large system can be quite large. Generally, the software development process is characterized by frequent recompilations in response to small incremental changes in the system, particularly during the debugging phase. If the entire system must be recompiled every time, this can become unmanageably burdensome. With good support for separate compilation, it should be possible to make arbitrary changes to the implementation of a module without requiring recompilation of other modules which import it, so long as no changes are made to its public interface.

Secondly, there is a desire to be able to use precompiled code from libraries without necessarily having access to the original source. Most programming these days involves interaction with at least a few code libraries, and while some of these will be in-house or public domain code, many will be proprietary. It is important to provide support for doing this in a reasonable way.

Finally, there is a certain amount of interest in being able to provide composable programs using modules. Many commercial programs are designed to run under vastly differing environments, and hence must either be designed to function under all of them, or else distributed with separate versions for different environments. So for example, in some environments memory is at a premium and should be conserved even at expense of performance, whereas in others the opposite is true. Precompiled modules provide an interesting take on this by allowing a single program to be made and distributed with several different implementations of a module, each satisfying different constraints. The user can then choose which modules get linked together to form the final program depending on local constraints. This is also useful for writing portable code. Current software distributions frequently isolate all operating system specific code into a single module so that specializing the distribution to a specific operating system is simply a matter of linking the system with the module for that particular operating system.

PolyTOIL already provides a certain amount of support for separate compilation vertically in the class hierarchy, since we do not need to recompile methods inherited from super-classes. This means that extensive class libraries could be used without the need to have the source code available. It is probable as well that classes could be used as a unit of separate compilation across the class hierarchy. However, we may in general wish to

provide separate compilation for more than just classes. This is especially true given that in **PolyTOIL** classes are first class values, and frequently appear as return values from functions. There are also issues concerning the granularity of the compilation units that need to be considered. In general, while smaller compilation units seem to support slightly more separate compilation, they tend to make significant sacrifices in terms of speed of compilation and the ability of the compiler to perform global optimizations without recompiling the entire system.

Perhaps a more fundamental problem is that classes are in a sense overspecified for separate compilation in that any significant change to a class implementation will almost certainly change its instance variables and hence its type. As a result, any code that deals directly with classes (even solely to the extent of instantiating them to objects) cannot be compiled in isolation from the class since classes are aggregates whose size depends on their implementation. This can be partially avoided by defining functions with the class to return an object instantiating the class, allowing most client code to avoid handling the class directly. Specific implementation details tend to be defined by the instance variables and hidden methods of a class, which do not appear in type of the objects generated by the class.

To the extent then that we can treat objects as abstract interfaces for the classes that they instantiate, this is not an overwhelming problem. However, objects must always contain all of the public methods of a class, and it is very frequently the case that a class *must* provide public methods returning all or part of its internal structure in order to implement other functionality with reasonable efficiency, as was the case in figure 4.1 above. In these situations, in general quite common with objects, the type of the object ends up indirectly depending on implementation specific details that would otherwise only appear in the class type. As a result, any significant change to the implementation of the class will almost certainly change the interface presented by the object, and hence will force recompilation of all other units in which the object appears. So for example in figure 4.1, `IntSetType` includes all of the list methods in its type in order to support efficient intersection, but consequently must reveal its implementation completely. In general then, while **PolyTOIL** certainly provides opportunity for separate compilation, it does not provide the abstraction necessary to make separate compilation as effective as it should be. It is of little use to be able to compile things separately if the interfaces from which inter-compilation unit dependencies are determined change whenever the implementation changes.

4.2 Module systems in existing languages

This problem is not at all specific to **PolyTOIL**. Most current object-oriented languages generally use classes as the only unit of modular structure and many seem to suffer from the same problems. In fact, two of the most common object-oriented languages, C++ and

Eiffel, suffer from this to an even greater degree. Eiffel has no real concept of a module beyond the class, and while most C++ compilers provide a little more through limited scoping rules between files, C++ provides weak modular abstraction at best, and does not strongly enforce what little it provides.

Both languages export instance variables as an explicit part of the class interface, and have no concept of a separate object type. The complete class definition *is* the interface in both languages. As a result, all clients of a class in either language are completely dependent (from a compilation standpoint) on the implementations of the classes. In C++, this is made even worse by the fact that classes are aggregates, not pointers, and hence must have information about their size hardwired into any client code (recall that in C++ classes serve as types for objects, and must specify the makeup of the object in complete detail) [ES90]. Client code must therefore either be written entirely using pointers, or else suffer from forced recompilation every time the size of a class changes. In the case of Eiffel, the usefulness of separate compilation is further compromised by the need (as yet unimplemented) for global link time checks to insure system validity, meaning that there is no guarantee that previously compiled code cannot later produce type errors [Mey92].

In general, the support for modular programming provided by these two languages does not sufficiently address the issues of programming in the large. Neither language provides any means of using modules to support type abstraction in any advanced sense. Eiffel does provide a means for creating generic classes, but types may not appear as members of classes (whereas most strong module systems provide this functionality). All current implementations of C++ provide a weak form of generics through the use of special macros called templates. In general, the behavior of templates is not well defined, frequently surprising even experienced C++ programmers. Moreover, because templates are simply defined as macros, they are not actually compiled until they are instantiated, resulting in both late detection of type errors, and enormous duplication of generated code. C++ suffers further from the lack of any language level conception of dependencies between the "header" files used as interfaces and their clients. Programmers must use programs like **Make** or special compilation environments to attempt to manually express dependencies between modular units.

4.2.1 SML - Transparent Types and Separate Compilation

The language **SML** provides one of the stronger module systems available. It is one of the very few which provide for parameterization of modules over modules, creating in essence a small typed "module language". Structures play the part of values, signatures that of types, and functors serve as functions from "values" to "values". Programmers can use these facilities to link together different modules in different ways. This allows large programs to be written as smaller composable units which can then be combined in different ways. Unfortunately though, signatures cannot completely represent the type of functor

```
signature ModuleSig =
    sig
        type t;
        val eq: t→t→bool
    end;

structure ModuleStruct: ModuleSig =
    struct
        type t = int;
        fun eq (a: t) (b: t) = (a = b);
    end;

val numEqual = ModuleStruct.eq 1 2;
```

Figure 4.2: Transparent types in SML

parameters, since the full structure of exported types is visible even though the type structure is not present in the signature. For instance, in Figure 4.2, the type of t is unspecified in *ModuleSig*, but is nonetheless visible outside of *ModuleStruct*, allowing the application of the `eq` function to two integers on the last line. Such "transparency" of types in ML means that the interface of a structure (its signature) is not sufficient for compilation to take place. The implementations of all structures in the system must be available at compile time. The only way to get around this is to make heavy use of functors, postponing the binding of structure until the end. This is not especially satisfactory, especially as it means that many type checking errors cannot be caught until link time. While this lack of support for separate compilation is consistent with **SML**'s intended use as an interactive prototyping language, it has proved to be a significant problem for people trying to build large systems in **SML**.[Ler94]

4.2.2 Modula-3 - a strong module system

Probably the strongest currently available module system is that found in the Modula family of languages - particularly Modula-3. Modula-3 provides excellent facilities for modular programming including several levels of type abstraction, the ability to write generic modules, and support for separate compilation. The Modula-3 module system consists of interfaces and modules. Modules contain declarations and executable code, while interfaces specify what declarations are exported by the corresponding module. Unlike **SML**, Modula-3 interfaces completely specify all information available to other modules, meaning

that modules which import other modules only need access to the interfaces of the imported modules. (The modules themselves can then be compiled in any order.) Moreover, changes in a module cannot force recompilation of other modules unless changes were made in the interface as well [Har92, Nel91]. Even more interesting from a language design standpoint are the facilities provided by Modula-3 for type abstraction.

Of the existing module systems, we were most impressed by that of Modula-3, and much of the $\mathcal{LOOM}$ module design is patterned after the Modula-3 module features (modified for a language without subtyping). We devoted a significant amount of study to the **SML** module system, and while in the end it had relatively few direct influences on the $\mathcal{LOOM}$ modules, it has provided a number of ideas for further work with modules in $\mathcal{LOOM}$ (see section 7.2). In general, the **SML** module system combines some quite powerful features with some very significant problems, and it is not yet clear how to retain the one without the other. We will examine Modula-3 more closely in the next chapter.

Chapter 5

Language Design

The most important part of adding modules to $\mathcal{LOOM}$ was making the decisions that defined the modules and their behaviors. Once these decisions were made, the actual process of implementing them, while challenging, was conceptually more straightforward. Making the decisions was frequently an agonizing process of analyzing existing systems and thinking about the ramifications of each choice. Many different ideas were examined and discarded. In general, we designed conservatively, leaving more complicated decisions to be examined in the light of the evaluation of the core system. However, care needed to be taken to make the core design flexible enough to incorporate later changes. We believe that to a large part we succeeded with this, and indeed, the module system has already gone through several modifications, with more planned. In the following sections, we present some of the issues that went into our choices of type checking and semantic rules, and then present some of the most important of these rules in the context of an example.

5.1 Abstraction - Partial, Complete, and Manifest Types

Most languages that provide some sort of modular type abstraction mechanism only provide support for at most two types of type revelation - completely opaque or completely transparent. We have already seen an example of transparent types in **SML** in section 4.2.1. While transparent types do cause problems with separate compilation, they also provide a functionality that is important: the ability to share type information between modules. If structural type equivalence is used, transparent types can allow for type equivalence judgments to be made between modules. This is very important in building up a system of modules each of which contains a different set of operations on a single type. The canonical **SML** example demonstrating the usefulness of transparent types is a token type shared between a lexical analyzer and a parser. Clearly there is a functionality here that is important.

Manifest Types

The real problem with the transparent types is not that they reveal too much about the types - indeed they are used in exactly those contexts where one wishes to reveal the implementation of the types. The problem is that they are revealed not by the interface, but by the implementation. One solution to this difficulty is to add this type information to the interface of the module as explicitly exported information. This allows the type information to be propagated without violating the integrity of the module abstraction. This kind of abstraction is is referred to as a *manifest type specification*, and has been advanced as a possible solution to the **SML** separate compilation problem [Ler94]. Modula-3 takes this approach to exporting complete type information from modules. An interface may contain declarations of the form `TYPE T = INTEGER, TYPE S = ...`, etc. Any type importing the module in which the declarations appear may use all of the information given by the type declaration. Note that this is distinct from transparent types in that the type information actually appears in the interface to the module and hence can be used without access to the implementation module.

Opaque Types

On the opposite end of the spectrum from manifest types are opaque, or fully abstract types. Opaque types are types that are exported abstractly - no information is given as to their nature, and all operations on the type must be defined within the scope of the module where the type is defined. Generally, opaque types behave as unique types, with no other type than themselves judged equivalent to them, even if there is a coincidental structural equivalence. Opaque types are typically used in programming abstract data types where specifics of the implementation are hidden from the client. For example in **SML**, an abstype declaration produces a type name about which no structural information is known, and for which the only operations are those defined within the scope of the abstype definition. Complete abstraction of types is a very important feature of a programming language since it allows modules to be written in such a way as to be arbitrarily replaceable. Moreover, by guaranteeing that the types are protected from even accidental equivalence with outside types, there is no way for client modules to accidentally or intentionally violate the conceptual abstraction of the module. This makes it much simpler to guarantee invariants, since the only way to create values of the opaque type is through the predefined operations. In the set example from figure 4.1, we could imagine making `IntSetType` an abstract type in a module, defining functions `intersect` and `remove` as functions defined in the module and exported from the interface.

The problem with opaque types is that they frequently provide a stronger abstraction than we might wish. In particular, there are situations where other modules may need to be able to perform certain operations on an abstract type that rely only on it having certain

general properties. In an object-oriented paradigm, this might be the property of having a `compare` method, or a `print` method associated with it. In general, anything which has a print method should be able to be told to print itself, and it seems cumbersome to have to write and export a function to print a completely abstract type, when all the print function does is send the print message to the object within the scope of the module. In the set example, this can be seen with the `intersect` and `remove` methods. An implementation of this as an opaque type would require explicit intersect and remove functions to be exported, the body of each of which would simply be a single message send. It is cumbersome to have to write functions to allow access to every method that we actually want to export, just to be able to hide the ones that we do not wish to export. Note too that methods like `print` and `compare` or `intersect` are such that their interface almost never changes, no matter what changes are made to the implementation of the underlying type. On the other hand, we may still not want to provide free access to the implementations, for all the reasons described in section 4.1.2. This suggests the need for something stronger than complete revelation, but more flexible than opaque types - in particular, this suggests the need for partial abstraction.

Translucent Types

Translucent types, or in general partial abstraction, is the idea of allowing some information to escape the abstraction barrier while keeping the essential implementation abstract. The most common means of doing this is to use subtyping to express the partial revelation - that is, the interface specifies that the abstract type is a subtype of some other simpler type, allowing elements of the abstract type to be treated as elements of the supertype. By choosing a supertype that allows only those operations which we wish to be publicly revealed, we can allow public access without violating abstraction. Returning to the set example, and ignoring for the moment the fact that in **PolyTOIL** `IntSetType` would have no proper subtypes[1], we could declare `IntSetType` to be a subtype of some abstract type `AbsIntSetType` which had only the methods `remove` and `intersect`. A client of this module would then have sufficient information to send `remove` and `intersect` messages to a set, but would have no information about other methods supported by the object. The programmer would also be free to replace the implementation of `IntSetType` with any other implementation that provided the same two methods without any change to the interface.

Modula-3 supports partial revelation for reference types through subtyping constraints placed on type declarations. What Modula-3 calls opaque types are declarations of the form `TYPE T <: U` where `U` is `REFANY`, the supertype of all reference types, or `ROOT`, the supertype of all object types. Partial abstraction can be achieved simply by providing a more specific type as the upper bound in the declaration. Somewhere in the program the

[1] Recall that the contravariant occurrence of **MyType** in the `intersect` method prevents `IntSetType` from having any non-trivial subtypes. (section 2.1)

complete type of `T` must be revealed - for every opaque type declaration, there must be exactly one complete revelation, but it may occur in other modules. A complete revelation is a declaration of the form `REVEAL T = V` where `V` must be a "branded" reference or object type expression. (Branding is simply a way of guaranteeing uniqueness of abstract types so that the typeCase and narrow facilities cannot be used to defeat the abstraction). In addition to this partial opacity, Modula-3 also provides a facility for subsequent partial revelations on an opaque or partially opaque type. This is denoted `REVEAL T <: W` and specifies that in the current scope, `T` may be treated as being "at least" `W`. In general there be any number of these partial revelations, with the restriction that all of the revealed supertypes be linearly ordered by the <: relation, and that for all such `Ws`, `V <: W` must hold [Nel91].

5.2 Designing the $\mathcal{LOOM}$ Modules

The process of designing programming language features involves a good deal of feedback between syntactic issues and semantic issues. It is not clear at all that the syntactic design can or should precede the semantic design of a language - rather it would seem that the two are intrinsically linked. Indeed, throughout the design of $\mathcal{LOOM}$, syntactic choices were primarily attempts to express the intended semantics in as intuitive a way as possible, and hence are in some ways a direct outgrowth of the semantics. On the other hand, the semantics themselves were designed with a certain syntactic elegance in mind. In general, while for the sake of clarity we introduce the $\mathcal{LOOM}$ module syntax before we attempt to describe the associated semantic properties, the reader should keep in mind that many of the syntactic decisions are best understood in the context of the intended semantic meaning. Throughout this section, we assume familiarity with the base $\mathcal{LOOM}$ language as described in chapter 2. The syntax, typing rules, and semantics of $\mathcal{LOOM}$ are given in entirety in appendices A, B and C.

5.2.1 Syntax issues

The syntax of the $\mathcal{LOOM}$ modules (figure 5.1) is a fairly straightforward extension of the original **PolyTOIL** syntax. Excluding the module syntax, the only significant syntactic change to the existing language is that identifiers can now be path names, where a path name is a simple identifier qualified by an interface name. The standard notation for path names is generally the module name, followed by a connective, followed by the element name. So for instance in **SML** a function `f` from structure `A` would be designated as `A.f`. For the $\mathcal{LOOM}$ modules we avoided the "." notation to avoid confusion with message sending, and instead borrowed the C++ scoping syntax, making the above `A::f`. This idea of qualification of names is one of the fundamental aspects of namespace management in modular programming: that clients of a module have access to elements of the module only

$$
\begin{array}{rcl}
\textit{Module} & ::= & \textit{Interface} \mid \textit{Implementation} \mid \textit{Program} \\
\textit{Interface} & ::= & \texttt{Interface}\ \langle\text{id}\rangle\ \textit{ImportList}\ \textit{DeclList}\ \texttt{end} \\
\textit{Implementation} & ::= & \texttt{Module Implements}\ \langle\text{id}\rangle\ \textit{ImportList};\ \texttt{Type}\ \textit{AbbrevList}\ \texttt{Const}\ \textit{ConstList}\ \texttt{end} \\
\textit{Program} & ::= & \texttt{Program}\ \langle\text{id}\rangle\ \textit{ImportList}\ \textit{AbbrevList}\ \textit{UnitBlock} \\
\textit{ImportList} & ::= & \texttt{Imports}\ \textit{IdList};\ \mid\ \emptyset \\
\textit{DeclList} & ::= & \textit{Revelation}; \textit{DeclList}\ \mid\ \emptyset \\
\textit{Revelation} & ::= & \textit{PartialRev}\ \mid\ \textit{Assertion} \\
\textit{PartialRev} & ::= & \langle\text{id}\rangle \langle\#\ \textit{TypeExp} \\
\textit{Assertion} & ::= & \langle\text{id}\rangle\ =\ \textit{TypeExp}\ \mid \\
 & ::= & \langle\text{id}\rangle\text{:}\ \textit{TypeExp} \\
\textit{ConstList} & ::= & \langle\text{id}\rangle\ =\ \textit{Expr}\text{:}\ \textit{TypeExp} \\
\textit{IdList} & ::= & \langle\text{id}\rangle; \textit{IdList}\ \mid\ \langle\text{id}\rangle \\
\textit{PathName} & ::= & < id > \mid < interfacename > :: < id >
\end{array}
$$

Figure 5.1: BNF grammar for $\mathcal{LOOM}$ modules

`Interface` B `imports` A;

$$\begin{array}{lcl} T & = & \textbf{INTEGER} \\ S & <\# & \textit{ObjectType}\ m\text{: } \textbf{proc}(A::T);\ \textbf{end} \\ f & : & \textbf{func}():T \end{array}$$

`end`

Figure 5.2: Example $\mathcal{LOOM}$ interface

through full path names, constraining the problem of avoiding name conflicts to within a single module.

In $\mathcal{LOOM}$, all paths are referenced relative to interfaces as opposed to implementation modules, unlike **SML** which uses the structure name to qualify identifiers. This reflects the fact that in **SML**, signatures play a role closer to types of accessible values, whereas in $\mathcal{LOOM}$, clients never deal with implementation modules directly. As a result, **SML** can allow multiple structures to implement a single signature, allowing parameterization over modules in a more powerful and syntactically elegant manner. Modula-3 supports similar functionality through generic modules parameterized over interfaces, but the result is a fairly cumbersome mechanism [Har92]. On the other hand, restricting modular access to interfaces makes it much easier to compose modules into programs without relying on specific implementations, since the name of the implementation of an interface need never appear anywhere else in the program. To get the same effect in **SML** requires the pervasive use of functors to bind specific implementation structures to general structure names used pervasively throughout the rest of the program. In $\mathcal{LOOM}$, we have chosen to restrict access to modules to be through interfaces. In fact, in the current design, there is no way to bind an implementation to a name. Implementations simply specify which interface they are implementing. In section 7.2 we will discuss possible extensions to the $\mathcal{LOOM}$ module system that would require the ability to name implementation modules, a relatively straightforward syntactic change.

The syntax for the module system itself is relatively simple. Most of the new syntax appears in interface modules, where it is necessary to allow new kinds of declarations not previously available in **PolyTOIL**. Figure 5.1 gives a BNF grammar for the module system. Words in boldface are language keywords. In general, Modules can be interfaces, implementations, or programs. Implementation modules specify an interface to implement, an optional list of interfaces to import, an optional list of type definitions, and an optional list of constant definitions. Program modules essentially retain the syntax of original **PolyTOIL** programs, with the addition of an import list. An example of an interface module appears in figure 5.2. Interface modules consist of a name, an import list, and a list of *revelations*. A revelation is an expression of partial or total information about the nature of an identifier

exported by the interface. The types of exported constants are always totally revealed - their full type is given in the interface. Indeed it is not clear what it would mean for a constant to be partially revealed. Note however that the type of an exported constant may include names of partially revealed types. Example 5.2 gives an example of a constant, f, exported in an interface. The other two revelations in the figure refer to types exported by the interface. Exported types may be fully revealed by specifying that they are equal to another type, as with type T from figure 5.2, or partially revealed by specifying that they match another type, as with S.

These revelation mechanisms provide an expressive way of specifying the properties of a module. The one remaining difficulty with the syntax arises with partially revealed parameterized types. Figure 5.3 shows such a revelation as it would appear in the old notation for parameterized types. The intent of this example is to define a type that when

$$U = \mathit{ObjectType}\ m\colon\ \mathbf{proc}();\ \mathbf{end}$$
$$S <\#\ \mathbf{TFunc}[T <\# U]\,\mathit{ObjectType}\ f\colon\ \mathbf{proc}(U);\ \mathbf{end}$$

Figure 5.3: Parameterized types with partial revelation: Old syntax

$$S[T <\# U] <\#\ \mathit{ObjectType}\ f\colon\ \mathbf{proc}(T);\mathbf{end}$$

Figure 5.4: Parameterized types with partial revelation: New syntax

instantiated as $S[\tau]$ will match any type of the form $\mathit{ObjectType}\ f\colon\ \mathbf{proc}(\tau);\ \mathbf{end}$. This intended semantics is not at all clear from the syntax which seems to imply a higher order matching relation defined over functions from types to types. To resolve this, the syntax for parameterized types changes in $\mathcal{LOOM}$ to that of figure 5.4. This syntax suggests the idea that the matching relationship is defined over instantiations of the type function, rather than the type function itself. This syntax change has not yet been implemented in the interpreter, but we do not expect significant difficulties in the transition.

5.2.2 Semantic issues

The semantic issues in the design of $\mathcal{LOOM}$ were the most interesting and difficult to deal with. It is still not clear what the best ways of handling some of these issues are, even where a general direction is visible. There are a significant number of tradeoffs involved in the choices made here, and it will be interesting to see how the choices made bear up under pressure of actual programs.

Transitivity of import

In **SML**, all structures occupy a global namespace. Any structure can reference elements of another structure simply by giving its full path name. In some ways, this is a nice feature in that it removes the burden of specifying for each structure what other structures it uses. In general though, this kind of system has some very undesirable results. Making the programmer explicitly list which modules are used by a module helps both to document dependencies between modules for clients and other programmers, and to express the module dependencies to the compiler. This makes separate compilation much easier, and more powerful, since the compiler can know *exactly* which modules are dependent on which others. We therefore decided to make modules completely opaque to modules that do not import them. This then raises the question of exactly how import should behave.

We say that import is transitive if when module `C` imports module `B` and module `B` imports module `A`, `C` has access to `A` as if it had imported it explicitly. A transitive module structure makes for shorter import lists, since it is not necessary to re-import modules imported by other modules. Note too that if a type `t` from module `A` appears in the type of a function `f` imported from `B`, `C` must import `A` either directly or indirectly in order to make use of `f`. It is quite common for types from modules to appear in several cascaded modules - consider a node module imported by a list module which is in turn imported by a client module. On the other hand, transitive import can end up bringing unwanted elements in with needed elements. Generally speaking, transitive import tends to clutter the namespace unless a selection mechanism is provided, while intransitive import tends to force ungainly import lists.

Our feeling in designing the modules was that transitive import is a more intuitive idea. It is very counter-intuitive to be able to import functions which have types which are not importable from the same module. For this reason, we chose to make imports transitive, with the intention of providing a mechanism for either selective import, or selective export. A selective import mechanism would be essentially a `from` list within an `export` list, allowing a client to specify exactly what things should be imported from the parent module. This is useful even in normal usage, since clients may not need everything provided by the parent anyway. An equally interesting possibility is to allow a module to specify which things from its imports it wishes to export. This is very nice because it removes the burden of finding out exactly what imports are necessary from the client module. The parent module specifies those things which a client will need in order to make use of it and the client simply does a complete import.

Export Rules and Type Equivalence

Another interesting design problem is the question of exactly what things should be exported from a module and how they should behave. This proved to be the most interesting and

challenging question of the design. The constraints of separate compilation require us to allow other modules to use only information presented in the interface, but the question remains of exactly what can then appear in the interface. Many languages, such as Modula-3 and Ada allow programmers to include blocks of executable code within the scope of the module, with the intention that it be run when the program starts executing. So for example in Modula-3, every module ends in a possibly empty block which performs initialization for the module. The module blocks are guaranteed to execute such that no module is initialized before any of its imports. (Since this refers only to implementation modules, the ordering is guaranteed to be acyclic) [Har92, Nel91]. Similarly, many languages also permit modules to contain runtime variables as members.

In general, we could not come up with any sufficiently convincing arguments for allowing variables or executable sections in modules that could not be easily addressed with objects. While there are a few problems that might benefit particularly from being able to hold state as part of a module, we felt that the added complexity was unwarranted at this stage. As a result, $\mathcal{LOOM}$ modules may contain and export constants and types only. In section 7.2 we discuss the possibility of adding some of this functionality.

Constant export is relatively straightforward, with the type of the constant appearing in the interface as it appears in the implementation module. In figure 5.2, the constant `f` is exported as a function that returns an element of type `T`. Similarly, the type `T` is being exported as the type `INTEGER`. Throughout the body of the implementation of the module, and within the scope of any import of `B`, `T` will be equivalent to the type integer. Notice that this use of types is essentially equivalent to the old **PolyTOIL** notion of type names as abbreviations for full structural types. This is not the case for translucent export.

The example on the second line of figure 5.2 is an example of the adaptation of the partial abstraction mechanism of Modula-3 to $\mathcal{LOOM}$, a language without subtyping. In this example, `S` is declared to be a type that matches *ObjectType* m: **proc**(A::T); **end**. Thus, any module importing this interface knows of the existence of `S`, and knows that `S` contains at least the method `m`. Clearly then, objects of type `S` may be sent the message `m`, and may be assigned to variables of declared type `S`. Less obviously, objects of type `S` may be assigned to variables of any type $\#\tau$ such that *ObjectType* m: **proc**(A::T); **end** $<\#\,\tau$. Semantically, the type `S` is defined to be unique: that is, outside the context of the module implementation, it may only be judged equivalent to itself.[2] In other words, we must use name equivalence rules for type equivalence judgments on translucent types since we have no way of knowing the actual type. However, a translucent type *can* be judged to *match* another type by transitivity since we have an upper bound on the type. Outside of the module, clients are aware of both the existence of `S` and of the existence and type of a subset of its methods and hence can operate on it in nontrivial ways independent of

[2]Note though that any type that is defined to *be* `S` in a manifest type specification may still be judged equivalent to `S`, since a manifest type specification is semantically equivalent to type abbreviation.

predefined operations within the body of the module. Within the implementation module, there must appear a complete definition of `S` of the form `S = W` for some type expression or type name `W`.[3] Within the scope of the module body, `S` may then be treated as a type identical to `W`, allowing structural equivalence to hold within the module scope.

This is a very important result, because it allows interaction between partially abstract types and the other abstraction mechanisms of $\mathcal{LOOM}$. In particular, translucent types can be used with forms of bounded quantification such as hash types and bounded polymorphism. The fact that we can make matching judgments based on the upper bound also allows for message sends to objects of the translucent type. Recall that message sends are type checked by the rule

$$MSG \qquad \frac{C, E \vdash o{:}\,\gamma, \quad C \vdash \gamma <\# \mathit{ObjectType}\{m{:}\,\tau\}}{C, E \vdash o \Leftarrow m{:}\,\tau[\gamma/\mathbf{MyType}]}$$

If o (from figure 5.2) has type B::S, then by the rule above, sending a message m to o requires only that we be able to prove that $B{::}\,S <\# \mathit{ObjectType}\ m{:}\ \mathbf{proc}(A{::}\,T);\ \mathbf{end}$. Given the upper bound on `S` from the interface, this is easy to show. As a result, any method name that appears in the upper bound of `S` may be sent to elements of type `S`.[4] This provides a great deal of flexibility to programmers.

In **PolyTOIL**, there is also a notion of stronger abstraction. **PolyTOIL** allows unbounded quantification of type parameters to functions, essentially providing the ability to write completely generic functions or methods. While it may not be immediately clear what purpose this serves, it is in fact fairly useful for defining functions that return classes implementing generic containers that do not require any functionality from their elements. In the original $\mathcal{LOOM}$ design, we included a form of opaque export based on this idea. This allowed types to be exported completely abstractly, so that the only operations defined on values of such a type were those defined within the scope of the module in which the type was defined. In general however, it does not seem that completely abstract types like this can be supported easily in the presence of true separate compilation, since the possibility that the actual type is a class[5] precludes the possibility of determining allocation needs based solely on module interfaces. While it is possible that this could be implicitly translated to indirect references heap allocated variables, this seems difficult at best. Modula-3 avoids this problem by supporting opaque types as subtypes of `REFANY` (see section 5.1). Client modules therefore need only allocate space for a pointer. While $\mathcal{LOOM}$ does not support pointers, recall that all objects inherit from (and hence match) the common superclass `Top`. This allows an essentially completely generic object type to be exported by

[3] This does not mean that the definition of `W` must be visible: it is possible that `W` is imported abstractly from another module.

[4] Note the difference here from an existentially quantified # type (section 2.2, section 2.3) to which even messages that appear in the supertype may be refused if **MyType** appears in the upper bound.

[5] The only direct (non-pointer) aggregate type in $\mathcal{LOOM}$

simply exporting it as matching `Top`. It would also be possible to simply restrict the use of completely opaque types to non-class types, but in general this seems rather arbitrary, and does not add much in the way of functionality.

Scoping/Naming Rules

An important component of the usefulness of modules is that they resolve namespace conflicts. As long as interfaces are guaranteed to be unique, anything defined in a module has a unique name, called its path name, composed of the name of the interface from which it is imported and the name given to it within the interface. This is very nice in that it both guarantees uniqueness of names and also allows names to carry information about their origin. However, these path names tend to be more cumbersome than simple names. To alleviate this somewhat, we permit elements to be denoted by their simple names within the scope of the interface and module in which they are defined. Since all other names must be path names, we still can never run into naming conflicts. Note though that as a result, types that contain simple names in an interface may appear differently when they are used in other modules where the full pathnames must appear. So for example on line three of figure 5.2, the constant f and its return type T appear as simple names, but any client module would have refer to f as B::f, and would perceive it as returning an element of type B::T. Eventually, we would like to consider adding facilities for importing elements of a parent with specific simple names. However, this raises a number of extremely non-trivial issues when combined with transitive import, particularly if we allow the client to choose names for the imported elements. For example, if a module gets a copy of the same type name from two different imports (transitively), one or both may be renamed. It is in general somewhat difficult to arrange things so that renamings remain transparent after the first abstraction. In general it is to be desired that no amount of renaming should ever render a type unable to be judged equivalent to another version of itself.

5.3 Modular Type Checking

In **PolyTOIL**, type checking essentially was a two stage process - inferring a type for an expression from the type of its sub-expressions, and checking that its type was a subtype of its declared type. This two stage process carries over into base language of $\mathcal{LOOM}$. Where the $\mathcal{LOOM}$ type-checking process diverges significantly is in type checking modules, where the additional level of abstraction brings in a corresponding additional level to the type checking. In $\mathcal{LOOM}$ then, the type-checking process may be viewed as a three stage process in which first the types of constants are inferred, then checked against their declared types, and finally used to help check that the module in which they are defined is consistent with the interface it purports to implement. The idea of course is to maintain a buffer between implementation modules and clients. Since all a client module has available to

it is the interface of its imports, it depends on the actual implementation of the module being consistent with the interface. So long as this true, the information in the interface is sufficient to guarantee that no run-time type errors will occur.

5.3.1 Definitions

More formally, we may think of a system of modules as being type checked in the context of a *type revelation system* that keeps track of type information shared between modules. Recall from the type checking rules for **PolyTOIL** [BSvG95] that we define two sets of relations, $\mathcal{C}$ and $\mathbf{E}$ to maintain information used during type checking. $\mathcal{C}$ and $\mathbf{E}$ are used for determining the context of a type. The following is a formal definition of $\mathcal{C}$, the type constraint system.

Definition 5.3.1 (Type Constraint System) *Relations of the form $\sigma <\# \tau$ where σ and τ are type expressions, are said to be* type constraints. *A* type constraint system *is defined as follows:*

1. *The empty set, ϵ, is a type constraint system. In general we will denote an empty type constraint system as $\mathcal{C}_o$.*
2. *If $\mathcal{C}$ is a type constraint system, τ is a type variable or a type of the form $ObjectType(Mytype)\sigma$, and $t <\# \tau$ is a type constraint such that the type variable t does not appear free in $\mathcal{C}$ or τ, then $C \cup \{t <\# \tau\}$ is a type constraint system.*

So, C keeps track of matching constraints placed on type variables in the current scope. E holds variable names and the types associated with them. The formal definition of E is as follows:

Definition 5.3.2 (Type Assignment) *A type assignment $\mathbf{E}$ is a finite set of associations between variables and type expressions of the form $x{:}\,\tau$, where each x is unique in $\mathbf{E}$. If the relation $x{:}\,\tau \in \mathbf{E}$, then we write $\mathbf{E}(x) = \tau$.*

It is also necessary in $\mathcal{LOOM}$ to maintain an equivalence table of explicit type namings, denoted ST. In general this simply holds the relation between names and the types associated with them, but note that it is not simply possible to expand out names fully at any point, particularly in the presence of recursive types. We define an equivalence table as follows:

Definition 5.3.3 (Equivalence Table) *Relations of the form $t = \sigma$ are* type equivalences. *An* equivalence table *is defined as such:*

1. *The empty set, ϵ, is an equivalence table. In general we will denote an empty equivalence table as ST_o.*

2. *If* ST *is an equivalence table,* τ *is a type,* t *is a type variable that does not occur in* ST*, and* $t = \tau$ *is a type equivalence, then* $\mathsf{ST} \cup \{t = \tau\}$ *is an equivalence table.*

This then gives us the ability to keep track of type assignments, type constraints, and type equivalences. We define a type revelation system, denoted $\mathcal{M}$, in terms of these constructs as follows:

Definition 5.3.4 (Type Revelation System) *For names e,t and types* τ *and* σ*, declarations of the form*

- $e\colon \sigma$
- $t <\!\# \tau$
- $t = \tau$

are said to be type revelations. *A* type revelation system $\mathcal{M}$ *is defined to be a triple* ($\mathcal{C}$,$\mathbf{E}$,ST)*, where* $\mathcal{C}$ *is a type constraint system,* $\mathbf{E}$ *is a type assignment, and* ST *is an equivalence table. We define functions* $\overset{\mathcal{C}}{\hookleftarrow}$, $\overset{\mathbf{E}}{\hookleftarrow}$, *and* $\overset{\mathsf{ST}}{\hookleftarrow}$ *as follows:*

1. *The triple* $\mathcal{M}_o = (\mathcal{C}_o, \mathbf{E}_o, \mathsf{ST}_o)$ *is a type revelation system.*
2. *If* $\mathcal{M} = (\mathcal{C}, \mathbf{E}, \mathsf{ST})$ *is a type revelation system,* $\mathcal{C}' = \mathcal{C} \cup \{t <\!\# \tau\}$ *is a type constraint system, and* $t \notin \mathsf{ST}$ *then* $\mathcal{M} \overset{\mathcal{C}}{\hookleftarrow} (t <\!\# \tau) = (\mathcal{C}', \mathbf{E}, \mathsf{ST})$*, and* $\mathcal{M}' = (\mathcal{C}', \mathbf{E}, \mathsf{ST})$ *is a type revelation system.*
3. *If* $\mathcal{M} = (\mathcal{C}, \mathbf{E}, \mathsf{ST})$ *is a type revelation system and* $\mathbf{E}' = \mathbf{E} \cup \{e\colon \tau\}$ *is a type assignment, then* $\mathcal{M} \overset{\mathbf{E}}{\hookleftarrow} (e\colon \tau) = (\mathcal{C}, \mathbf{E}', \mathsf{ST})$*, and* $\mathcal{M}' = (\mathcal{C}, \mathbf{E}', \mathsf{ST})$ *is a type revelation system.*
4. *If* $\mathcal{M} = (\mathcal{C}, \mathbf{E}, \mathsf{ST})$ *is a type revelation system,* $\mathsf{ST}' = \mathsf{ST} \cup \{t = \tau\}$ *is an equivalence table, and* $t \notin \mathcal{C}$ *then* $\mathcal{M} \overset{\mathsf{ST}}{\hookleftarrow} (t = \tau) = (\mathcal{C}, \mathbf{E}, \mathsf{ST}')$*, and* $\mathcal{M}' = (\mathcal{C}, \mathbf{E}, \mathsf{ST}')$ *is a type revelation system.*

Interface modules generate type revelation systems which can be used by other interface and implementation modules to define the initial contexts in which their internal bodies are to be type checked. When an interface module is type checked, it generates two type revelation systems - one corresponding to information which is revealed to a client module, generally notated $\mathcal{M}^e$, and one corresponding to information which is revealed only to modules which wish to provide implementations for the interface, generally notated $\mathcal{M}^i$. When we are processing a list of declarations, we use the notation $decLst \overset{\mathbf{m}}{\rightsquigarrow} (\mathcal{M}^e, \mathcal{M}^i)$ to indicate that the declarations in *decLst reveal* the information in $\mathcal{M}^e$ and $\mathcal{M}^i$.

In a system with complete transitive export, this distinction becomes essentially syntactic - the only difference between the revelation system seen by a client and that seen

by the interface's implementation is in the names to which the elements within the system are bound. Client modules import contexts defined using full path names, whereas the implementation of an interface may use the unqualified names of elements defined in the implementation. In the current implementation, which does not support transitive imports, there is a more significant distinction in that the internal type revelation system provides access to all revelations imported in the interface, while the export revelation system only contains information revealed in the current interface. While this may at first seem somewhat arbitrary, there is in fact a logic behind this. For an interface C, the implementation of C will need access to at least those things imported by C, since it must implement everything in the interface. A client on the other hand, may be interested in only a subset of the functionality provided by C, and hence may only need access to a subset of the modules imported by C. So for instance, if D imports from C, but only uses m from C where the type of m does not mention B, then D will not have any use for the things which C gets from B, presumably used in other elements of C in which D is not interested.

The pairs of revelation systems generated by interfaces are associated with the name of the interface generating them in an *interface system.* An interface system contains triples representing all of the type revelation systems that have been generated so far.

Definition 5.3.5 (Interface System) *An* interface system $\mathcal{I}$ *is a finite set of associations between interface names and type revelation systems of the form* $(A, \mathcal{M}^e, \mathcal{M}^i)$*, where* A *is unique in* $\mathcal{I}$*. If the relation* $(A, \mathcal{M}^e, \mathcal{M}^i) \in \mathcal{I}$*, then we write* $\mathcal{I}(A) = (\mathcal{M}^e, \mathcal{M}^i)$*, and say* A reveals $(\mathcal{M}^e, \mathcal{M}^i)$ *given* $\mathcal{I}$*.*

$\mathcal{I}$ holds all of the information available to a module for import. Note that the only information that crosses module boundaries passes through $\mathcal{I}$, and that only interface modules may change the contents of $\mathcal{I}$. This is an important property to maintain for the sake of separate compilation, since as long as this is true no module can take advantage of information about the details of an interface and must rely solely on the information provided by the interface.

The last definition that we need before examining the type checking rules is necessary to support the distinction between externally visible path names and the simple names available only within interfaces and their implementations.

Definition 5.3.6 ($\mathsf{R}_\mathsf{A}(\tau, \mathcal{S})$) *For a type* τ*, an interface name* A*, and a set of type names (defined in* A*)* $\mathcal{S}$*, define* $\mathsf{R}_\mathsf{A}(,)$ *to be a function such that*

$$\mathsf{R}_\mathsf{A}(\tau, \mathcal{S}) = \begin{cases} \tau & \mathcal{S} = \epsilon \\ \tau'[A::t/t] & \text{if } t \in \mathcal{S} \text{ and } \tau' = \mathsf{R}_\mathsf{A}(\tau, \mathcal{S} - \{\mathbf{t}\}) \end{cases}$$

So for at any given time during the type checking of an interface A, $\mathcal{S}$ contains the set of simple names revealed by the module up to the current revelations, and hence $\mathsf{R}_\mathsf{A}(\tau, \mathcal{S})$ converts all unqualified occurrences of names t defined in A (listed in $\mathcal{S}$) in τ into names of the

$\mathcal{M}^e_B = \{$
$B::T = \mathbf{INT};$
$B::S <\# \mathit{ObjectType}\ m:\ \mathbf{proc}(A::T);\ \mathbf{end};$
$B::f:\mathbf{func}():B::T;$
$\}$
and
$\mathcal{M}^i_B = \{$
$T = \mathbf{INT};$
$S <\# \mathit{ObjectType}\ m:\ \mathbf{proc}(A::T);\ \mathbf{end};$
$f:\mathbf{func}():T;$
$A::T = \ldots$
$\}$

Figure 5.5: $\mathcal{M}^e_B, \mathcal{M}^i_B$ after type checking interface B

form A::t. Intuitively, $\mathsf{R}_\mathsf{A}(,)$ is converting types written using the module's private notation into types that use only the path names that are available to client modules. Without this conversion, general typing judgments about exported types would be impossible, since no information would be available about the names occurring within the types.

5.3.2 Modular type assignment rules and axioms

In this section, we present an overview of the type checking rules for $\mathcal{LOOM}$ modules. The reader should refer to the full listing of the rules in appendix B.3 for a complete listing of the rules, and for reference during the discussion of some of the more interesting cases that follows. Note that for the sake of simplicity, the rules presented here assume that a module imports only one other module. In general this does not significantly affect the rules, since adding multiple imports (as is supported by the interpreter) is simply a matter of performing an extra step to acquire the union of the sets of imported elements. We begin with an examination of the most important rules for type checking interfaces.

Recall from the previous section that the rule for type checking an interface module is given as follows:

$$\mathit{ModLst}:\mathit{Int} \qquad \frac{\begin{array}{c}\mathcal{S}_o, \mathcal{M}_o, \mathcal{M}^e_B \vdash \mathit{DecLst} \overset{\mathbf{m}}{\rightsquigarrow} (\mathcal{M}^e_A, \mathcal{M}^i_A) \\ \mathcal{I} \cup \{(A, \mathcal{M}^e_A, \mathcal{M}^i_A)\}, \mathbf{E} \vdash \mathit{ModLst}:\mathcal{I}'\end{array}}{\mathcal{I}, \mathbf{E} \vdash \mathit{Interface\ A\ Import\ B\ DecLst;\ ModLst}:\mathcal{I}'}$$

where $\mathcal{I}(B) = (\mathcal{M}^e_B, \mathcal{M}^i_B)$

For those unfamiliar with reading inference rules of this sort, the idea is to view them as a sort of upside-down proof tree. The section below the line represents the conclusion - in words here that under the assumptions of $\mathcal{I}$ and $\mathbf{E}$, an interface as specified followed by a possibly empty list of modules results in the interface system $\mathcal{I}'$. Above the line are the conditions necessary to prove the conclusion. Recall that $\mathcal{M}_o$ is an initial (empty) revelation system, and $\mathcal{S}_o$ is an initial (empty) set of type names. $\mathcal{M}^e_B$ is the exported type revelation system associated with B. So in the top line, we get the interface systems revealed by the declaration list - i.e. we use the rules associated with $(DecLst)$ to create two new revelation systems, $\mathcal{M}^e_A$ and $\mathcal{M}^i_A$. Recall that $\mathcal{M}^e_A$ represents the revelation system available to clients of A, while $\mathcal{M}^i_A$ represents the revelation system available to the implementor of A. These two revelation systems then get associated with A in $\mathcal{I}$ to type check the remainder of the modules. So for example, after type checking the interface presented in figure 5.2, any subsequent modules would be type checked with the addition of the association $(B, \mathcal{M}^e_B, \mathcal{M}^i_B)$ to $\mathcal{I}$ where $\mathcal{M}^e_B$ and $\mathcal{M}^i_B$ are as defined in figure 5.5. Note in particular the addition of the qualification to the names exported in $\mathcal{M}^e_B$.

The top line of the previous example relied on the rules for lists of declarations in order to generate the revelation systems to be placed into $\mathcal{I}$. An example of such a rule is the following, which applies to partial revelations of the form $t <\# \tau$.

$$(DecL{:}\ <\#) \qquad \frac{\mathcal{S} \cup \{t\}, \mathcal{M}^e_B{}', \mathcal{M}^i_B{}' \vdash DecLst \overset{\mathbf{m}}{\rightsquigarrow} (\mathcal{M}^e_B{}'', \mathcal{M}^i_B{}'')}{\mathcal{S}, \mathcal{M}^e_B, \mathcal{M}^i_B \vdash t <\# \tau;\ DecLst \overset{\mathbf{m}}{\rightsquigarrow} (\mathcal{M}^e_B{}'', \mathcal{M}^i_B{}'')}$$

where $\mathcal{M}^e_B{}' = \mathcal{M}^e_B \overset{\mathcal{C}}{\hookleftarrow} (\mathrm{B{::}t} <\# \mathsf{R_B}(\tau, \mathcal{S}))$ and $\mathcal{M}^i_B{}' = \mathcal{M}^i_B \overset{\mathcal{C}}{\hookleftarrow} (t <\# \tau)$
This rule simply continues checking the rest of the declarations with the addition of the name of the defined type to the list of names being replaced, and the injection of the matching constraint into the revelation systems. Note that before being injected into $\mathcal{M}^e_B$, the constraint is converted so that all simple names are replaced by path names. Continuing the example from above, the constraint $S <\#$ *ObjectType* m: **proc**$(A::T)$; **end**; would be added to $\mathcal{M}^e_B$, and the constraint $B::S <\#$ *ObjectType* m: **proc**$(A::T)$; **end**; would be added to $\mathcal{M}^i_B$. Finally, note that in the degenerate case when the declaration list is empty (i.e. all the declarations have been processed), the following rule applies -

$$(DecL{:}\emptyset) \qquad \mathcal{S}, \mathcal{M}^e_B, \mathcal{M}^i_B \vdash emptyDecLst \overset{\mathbf{m}}{\rightsquigarrow} (\mathcal{M}^e_B, \mathcal{M}^i_B)$$

The revelation systems that get returned therefore contain all of the revelation information presented in the interface.

The next example we will consider is that of type checking an implementation module. To put this in context, we will consider a few rules as they apply to an example implementation module (figure 5.6) corresponding to the interface given in figure 5.2. Note that S has been fully defined and an implementation has been provided for f. Also note that there

```
Module Implements B Imports C;
```

$$S = \mathit{ObjectType}\ m\text{: } \mathbf{proc}(A\text{::}T);\ \ldots\ \mathbf{end}$$
$$f = \mathbf{function}()\text{:}T\ \mathbf{begin}\ldots\mathbf{end}$$

```
end
```

Figure 5.6: Example $\mathcal{LOOM}$ module

is no need to re-import A since the elements of A are imported through the interface. This module is type checked by the following rule:

$$(\mathit{ModLst}\text{:}\mathit{Imp})\qquad \frac{\begin{array}{c}\mathcal{I}(A) = (\mathcal{M}^e_A, (\mathcal{C}_A, \mathbf{E}_A, \mathsf{ST}_A)) \\ \mathcal{I}(B) = ((\mathcal{C}_B, \mathbf{E}_B, \mathsf{ST}_B), \mathcal{M}^i_B) \\ \mathsf{ST}_B, \mathcal{C}_B \vdash \mathit{DefList} <\!\!* \ (\mathcal{C}_A, \mathbf{E}_A, \mathsf{ST}_A) \\ \mathcal{C}_B, \mathbf{E}_B, \mathsf{ST}_B \cup \mathsf{ST}_A \vdash \mathit{DefLst},\ \ \mathcal{I}, \mathbf{E} \vdash \mathit{rest}\text{:}\mathcal{I}'\end{array}}{\mathcal{I}, \mathbf{E} \vdash \mathit{Module\ Implements\ A\ Import\ B\ DefLst;\ rest}\text{:}\mathcal{I}'}$$

This is a fairly complicated rule, but in general the idea behind it is fairly straightforward. The first two lines extract the publicly exported revelations of C and the private revelations of B from $\mathcal{I}$. (Recall that by definition, a type revelation system is simply a triple of a type context, a type assignment, and an equivalence table). The list of definitions in the implementation are then checked against the privately exported revelation system of interface B to ensure that their declared types are consistent with their types declared in B, using the $<\!\!*$ relation discussed further below. Finally, the definition list is type checked using the normal type checking algorithm under the assumptions given in the public revelation system of C combined with any manifest types defined in interface B.

The actual type checking of the definition list is not especially interesting as it follows the standard algorithm for type checking constants in **PolyTOIL** or $\mathcal{LOOM}$ as given in appendix B. However, the process of determining whether or not a definition list is consistent with an interface is worth examining in more detail, as this a relatively new concept to $\mathcal{LOOM}$.

The $<\!\!*$ relation is very similar in many ways to the idea of matching. Indeed, languages such as **SML** allow a kind of matching on the functor level by allowing signature matching to occur with parameters - that is, structure parameters to a functor need only match, or in our terminology, be consistent with the signature specifying the parameter. The essential idea is that as with matching, the relation holds when the lower bound of the constraint holds a superset of the elements of the upper bound. In this case, this means that the module must provide implementations for at least those constants and types declared but not defined in the revelation system of the interface it wishes to implement. Furthermore, the types of the implementations must be either equal to their declared types, or in the case

of partially revealed types, must match the upper bound placed on them in the interface. An example of the rules for definition lists is the following, which deals with the definition of a type declared partially abstract in an interface.

$$(<\!*:<\!\#) \qquad \frac{(t <\!\# \tau') \in \mathcal{C}_A, \quad \mathcal{C}, \mathsf{ST} \vdash \tau <\!\# \tau' \quad \mathcal{C}, \mathsf{ST} \cup \{t = \tau\} \vdash \mathit{rest} <\!* (\mathcal{C}_A - \{t <\!\# \tau'\}, \mathbf{E}_A, \mathsf{ST}_A)}{\mathcal{C}, \mathsf{ST} \vdash t = \tau; \mathit{rest} <\!* (\mathcal{C}_A, \mathbf{E}_A, \mathsf{ST}_A)}$$

Using the example from figures 5.2 and 5.6, this rule could be applied to check if the definition $S =$ *ObjectType* m: **proc**$(A::T)$; ... **end** is consistent with the partial revelation declared in the interface B - $S <\!\#$ *ObjectType* m: **proc**$(A::T)$; **end**. Type checking of the interface module injects the partial revelation above into $\mathcal{C}_B$, where it can be found during the checking of $<\!*$. A matching query is made with the new defined type of S and the upper bound on the partial relation, *ObjectType* m: **proc**$(A::T)$; **end**. If the query succeeds, then the rest of the definitions may be checked against the revelation system with the partial revelation of S removed. Note that it is important to remove revelations from the revelation system as we discover that they are consistent, since we also require for consistency that every revelation in the interface (except manifest types, which are themselves definitions) be implemented in the implementation module. This is checked by the rule for an empty definition list, signifying that all of the definitions have been processed.

$$(<\!*:\emptyset) \qquad \mathcal{C}, \mathsf{ST} \vdash \mathit{emptyDefLst} <\!* (\{\}, \{\}, \mathsf{ST}_A)$$

Note that in fact this rule is something of a simplification, since the internal revelation system of an interface may also include imported values which do not get implemented in the implementation module. It is quite straightforward to check that all remaining revelations in a revelation system were imported into the interface, since revelations from the interface itself will always be simple names whereas revelations from imported modules must be qualified path names.

5.4 Modular Semantics

In this section, we provide an overview of the semantic rules for the $\mathcal{LOOM}$ modules, beginning with some definitions and continuing with a discussion of a few of the more interesting rules in detail. The reader should refer to appendix C.2 for a complete listing of the rules.

The semantic rules for the modules are significantly simpler than the type checking rules, reflecting the fact that a significant amount of the complexity of the module system lies in the powerful type abstraction mechanisms built into it. This simplicity is somewhat artificial however, due to the fact that we deal only with an interpreted language, and assume that all modules have been implemented before they are imported. These restrictions are highly

artificial, and result in a much simpler semantics. The tedious complexities of separate compilation are largely avoided in the interpreter.

In the $\mathcal{LOOM}$ interpreter, we largely ignore module interfaces, using them solely as a means of specifying what values escape the closed environments of implementation modules. A system of modules is evaluated in the context of a set of associations that defines the environments or interface name lists available for use by other modules. In the natural semantics in the appendices, we define an environment as such:

Definition 5.4.1 (Environment) *An environment ρ contains bindings of identifiers to values, such as closures, locations, classes etc. The notation $\rho[v/x]$ denotes the binding of value v to identifier f. If identifier f is bound in ρ to value v, we write $\rho(x) = v$.*

Environments provide the bindings of identifiers to values for lookup when they are encountered. Environments get carefully scoped and controlled so that correct local definitions of identifiers are maintained. This is particularly important with functions and classes, which need to maintain their own environments representing the local scope of their definition, since in general with first class functions and classes, there is no guarantee that bindings in the evaluation environment will be consistent with the bindings in the definition environment.

Definition 5.4.2 (Environment System) *An environment system Σ is a finite set of associations between triples of names, sets of exported names, and environments of the form $(A, \mathcal{N}, \rho)$, where A is unique in Σ. If the relation $(A, \mathcal{N}, \rho) \in \Sigma$, then we say $\Sigma(A) = (\mathcal{N}, \rho)$.*

Intuitively, Σ keeps track of all the interfaces that have been seen, along with both a set of identifiers indicating the values to be exported from the implementation and an environment that gets defined by the implementation to hold the exported values. Since we assume that modules will be implemented before they are imported, we do not have to deal with the rather complicated issues that arise in trying to build environments containing information that is not yet available.

Since we are primarily concerned with the behavior of the language at run time in the semantics, it is unsurprising that the most interesting rules deal with implementation modules which hold run time information, as opposed to interfaces which hold type checking information. The semantic behavior of interface modules consists of adding an entry to Σ containing the list of exported names and an initial environment containing pervasive values such as ground type operations and values imported through the interface. Note that while we do restrict export of constants from a module to those declared in interfaces, this is in principle not required since the type checking process guarantees that no illicit use of unexported information can be made.

The first rule we will look at describes the behavior of implementation modules. While in general we do not provide any facilities for including code in a module to be executed upon

creation, classes may evaluate initial values for instance variables when they themselves are evaluated, resulting in state changes. Therefore, we must maintain state throughout the evaluation of the module list.

$$ModLst{:}\,Imp \qquad \frac{\begin{array}{c}\Sigma(A) = (\mathcal{N}_A, \rho_A), \quad \Sigma(B) = (\mathcal{N}_B, \rho_B) \\ (defLst, \rho_A \cup \rho_B, \mathcal{N}, \emptyset, s) \downarrow^d (\rho', s') \\ (ModLst, (\Sigma - \{(A, \mathcal{N}_A, \rho_A)\}) \cup \{(A, \mathcal{N}_A, \rho')\}, \rho_o, s') \downarrow^\Sigma (\Sigma', s'')\end{array}}{(Module\ Implements\ A\ Import\ B\ defList; ModLst, \Sigma, \rho_o, s) \downarrow^\Sigma (\Sigma', s')}$$

In the rule above, the values for $\mathcal{N}_A$, ρ_A and ρ_B are initially extracted from the interface being implemented and the interface being imported. $\mathcal{N}_A$ contains the names of the values to be exported from A, ρ_A contains the initial environment in which to evaluate the definitions, and ρ_B contains the environment exported by interface B. Note that this relies on the fact that B itself must have already been implemented, since otherwise ρ_B will not contain Bs exports. The definition list itself is evaluated in the initial environment $\rho_A \cup \rho_B$, containing imports to the interface and to the module itself, as well as the pervasive values mentioned above. The definition list is also initially evaluated with the name list $\mathcal{N}_A$ extracted from the interface A, and an extra empty environment to which will be added bindings for all names in $\mathcal{N}_A$. Finally, the rest of the module list is evaluated with an environment system in which the original association $(A, \mathcal{N}_A, \rho_A)$ is replaced with the association $(A, \mathcal{N}_A, \rho')$, where ρ' is the environment created by binding each identifier appearing in $\mathcal{N}_A$ to its appropriate value from the list of definitions.

This last behavior is described by the rules for evaluating definition list. The rule below describes the behavior of an exported constant definition in a definition list.

$$DefLst \qquad \frac{\begin{array}{c}f \in \mathcal{N} \\ (e, \rho_i, s) \downarrow (v, s') \\ (DefLst, \rho_i[v/f], \mathcal{N}, \rho_e[v/A::f], s') \downarrow^d (\rho_e', s'')\end{array}}{((f = e{:}\,\tau; DefLst), \rho_i, \mathcal{N}, \rho_e, s) \downarrow^d (\rho_e', s'')}$$

The statement $f \in \mathcal{N}$ establishes that f is in fact exported. A simpler rule handles cases where $f \notin \mathcal{N}$ - that is, where f is not exported. The expression e is evaluated in the internal environment which contains all of the imported and recently defined values, and the resulting value v is then bound in both the internal and external environments. In the internal environment, it is bound to the simple name f - in the exported environment being built, it is bound to the name $A::f$. For the exported environment, the result is that $A::f$ is bound in an external environment, but was evaluated in an internal environment. Consequently, any closure information held in v will refer to the internal environment, allowing full access to the module. At the same time, nothing that does not appear in the interface gets exported to the new environment.

Chapter 6

The Interpreter

6.1 Implementation issues

A prototype type checker and interpreter for the language $\mathcal{LOOM}$ has been implemented in **SML**, using the sml-yacc/sml-lex tools. The code for the base language is modified from the code for the **PolyTOIL** [BSvG95] interpreter originally implemented by Robert van Gent [vG93] and Angela Schuett [Sch94], and subsequently worked on by Jasper Rosenberg and myself. Interestingly, almost all of the modifications for handling the hash types occur only in the type checking phase, with the interpreter remaining relatively unchanged. The module code on the other hand is split fairly evenly between the interpreter and the type checker and represents a more significant departure from the original interpreter.

The current implementation of the module interpreter is not overly complicated, relatively speaking. Note though that this implementation does not yet support transitive import or multiple interfaces, two language features that greatly complicate the handling of the type checking and the interpretation, especially if a means is provided to rename imported features. More significantly however, the current interpreter does not do any sort of separate compilation.

In the context of an interpreter, it is not completely clear what separate "compilation" means, but at the least, we would like to be able allow modules to be interpreted before the modules that they inherit. While the language design itself does not prohibit this, we have for the sake of simplicity allowed the prototype interpreter to assume that no interface will be imported before it is implemented by an implementation module. In general, interpreting modules without having already interpreted their imports is quite complicated. Each module must keep track of imported identifiers for which it does not yet have bindings so that when the unresolved references are finally defined, the interpreter may go back and patch in bindings wherever they are needed. Since these identifiers may need to be rebound within a module in many different places (within every closure generated in

the module, for example), this gets very complicated.

Overall, we are very satisfied with the interpreter as a tool for testing and evaluating the language. We have ported many of the programs written in **PolyTOIL** to $\mathcal{LOOM}$, and have begun the process of implementing the interpreter itself in $\mathcal{LOOM}$. It is important to begin testing $\mathcal{LOOM}$ on larger and larger programs, particularly with the modular extensions since they are specifically aimed at programming in the large. Currently however, running large programs through the interpreter is tedious because of the inherent inefficiency of the interpretation. For the time being, we have been making some efforts to replace the more inefficient sections of the interpreter with better implementations, but it is to be hoped that some time in the not too distant future we will be able to write a compiler for $\mathcal{LOOM}$. This would allow us to test much more fully the facilities that $\mathcal{LOOM}$ provides for separate compilation and programming in the large in general.

6.2 Using the interpreter

Readers interested in obtaining the $\mathcal{LOOM}$ interpreter should contact Professor Kim Bruce at Williams College. It can generally be made available through ftp services or the world wide web. The interpreter for $\mathcal{LOOM}$ is distributed in a directory called "interpreter". Before the main interpreter can be run, the files *grammar.sml* and *scanner.sml* be built. If these files are not present in the distribution directory, type *sml-yacc grammar* followed by *sml-lex scanner*. This should build the parser and scanner, assuming that there both the sml-yacc and sml-lex compiler generator tools are available on the destination system. The rest of the interpreter can be built in the **SML** compiler. Note that this system was implemented in Standard ML of New Jersey, version 0.93. We have not had access to the newest version of the compiler, but we have had a report that the interpreter did not compile successfully under a beta version of **SML** 1.09.

The **SML** compiler is interactive. To run the interpreter interactively, start **SML** and enter *use "loader.sml";*. This will start the compilation process. Compilation takes from 5 to 20 minutes, depending primarily on the memory available to the compiler. When the compilation is completed, a prompt will appear. To interpret a $\mathcal{LOOM}$ program named *program.loom*, enter *neval "program.loom";* at the prompt. To interpret the same program with the input file *input.loom*, enter *eval "program.loom" "input.loom";* at the prompt. An executable version of the interpreter can be built by using the file *make_exec.sml* instead of *loader.sml*. The executable version handles all normal unix input/output redirection. It must be called with a $\mathcal{LOOM}$ program as the first argument, but the input can come from the standard input or a file. If a second parameter is given, it is assumed to be an input file.

Chapter 7

Evaluation of the Language

7.1 Programming in $\mathcal{LOOM}$

One of the most important parts of evaluating the design of a language is writing programs in it. While $\mathcal{LOOM}$ has not been in existence long enough to have accumulated a large body of code, we have been very pleased with the results that we have seen so far. In general, the $\mathcal{LOOM}$ module structure seems to provide some very nice functionality that was not available in the original $\mathcal{LOOM}$ implementation. As an example we will reconsider the example of an efficient implementation of sets from figure 4.1, this time using the modular facilities added to $\mathcal{LOOM}$.

7.1.1 Levels of Access

Recall from section 4.1.2 that in attempting to provide an efficient destructive intersect method in a set class, we ran up against what seems to be a fundamental conflict between the need to be able to use knowledge of the implementation of the set to make intersection efficient, and the need to hide all knowledge of the implementation from clients for reasons of abstraction and separate compilation. So for example, we may implement sets using ordered lists or bit fields and in general would like both implementations to provide the same interface so that they could be interchanged freely. As can be seen in figure 4.1 this cannot be done well in **PolyTOIL** or the original $\mathcal{LOOM}$. The binary method `intersect` must be able to access its parameter `other` as an ordered list in order to do an efficient traversal implementing the intersection. Since in ordinary $\mathcal{LOOM}$ there is only one level of abstraction available, this means that in general any client of the set class may also access the set as a list, which violates abstraction and prohibits separate compilations.

Now consider the new implementation of the set class in figures 7.1 and 7.2, designed using the modular facilities of $\mathcal{LOOM}$. With modules, we get an extra level of abstraction to work with - that is, abstraction on the object level and abstraction on the module level.

```
Interface IntOrdList;

   OrdListType = ObjectType
      first: proc();
      next: proc();
      off: func():Boolean; -- is current elt off end of list?
      add: proc(Integer);
      deleteCur: proc();  -- current is next elt after deleteCur
      contains: func(Integer):Boolean;
      getCur: func():Integer
   end;

   OrdListClassType = ClassType ... end;

   OrdListClass: OrdListClassType;

end;

Interface SetOfInt;

   IntSetType <# ObjectType
      add: proc(Integer);
      remove: proc(Integer);
      contains: func(Integer):Boolean;
      intersect: proc(MyType)
   end;

   function newSet(): IntSetType;

end; -- Interface SetOfInt
```

Figure 7.1: Interface for sets with efficient intersection. Note that nothing in interface `SetOfInt` specifies any relation to or dependence on `IntOrdList`

```
Module Implements SetOfInt import IntOrdList;
type

   IntSetType = ObjectType include IntOrdList::OrdListType
      remove :proc(Integer);
      intersect: proc(MyType)
   end;

   ListSetClassType = ClassType include IntOrdList::OrdListClassType;
      methods visible
         remove :proc(Integer);
         intersect: proc(MyType)
      end;

const

   ListSetClass = class inherit IntOrdList::OrdListClass
      methods visible


            ⋮
        As in figure 4.1
            ⋮

         end;  -- class

   function newSet():IntSetType;
      begin
         return new(ListSetClass)
      end;

end -- Module
```

Figure 7.2: Module implementing sets with efficient intersection. Note how `ListSetClass` (defined in full in figure 4.1) takes advantage of the knowledge of the list implementation to do an intersect in a single pass across the two sets.

By providing different amounts of access to implementation details on the different levels, we can design a system that supports the functionality we need while maintaining the needed abstraction barrier between the module and its clients.

In figure 7.1, interface `SetOfInt` partially reveals the type of IntSet objects, and provides a function for creating a new set. Note that this last is necessary because we do not export the class from the implementation - in addition to being unnecessary from the client's perspective, the class reveals too much of the implementation. A client can therefore create and store objects of type `IntSetType`. More importantly, it has sufficient information to be able to send the messages `add`, `remove`, `contains` and `intersect`, but does not have any information about the other methods supported by `IntSetType`. As a result, the implementation of `IntSetType` may be changed to any type which supports the revealed methods of IntSetType without requiring any changes or recompilation to clients of the module. For example, the module in 7.2 implementing `SetOfInt` as an ordered list could be replaced with a new module using bit fields. Even more generally, a program could be distributed with both modules available, allowing the end user to choose which set implementation should be used based on local constraints.

In the implementation module, given in figure 7.2, a complete instantiation of `IntSetType` is provided - in this case using ordered lists. Within the scope of the module, the programmer is free to use this implementation specific information in the intersect method to get at the internal structure of the `other` parameter and perform efficient intersection. In this case, intersect takes advantage of the fact that the sets are implemented as ordered lists to do the entire intersect in a single pass across the two lists. An intersection performed using only the exported methods of `ListSetClass` would have to be done through repeated `contains` queries, changing the complexity from additive in the size of the sets to multiplicative in the size of the sets.

One might initially think about getting similar functionality without modules by defining a function `newSet2` as in figure 7.3. This fails for two reasons. First, the binary message `intersect` can never be sent to elements of type `#IntSetType2`, since in general methods whose types contain contravariant occurrences of **MyType** can never be sent to hash types. (Note that in figure 7.1, even though a client does not have definitive information about the true type of objects of type `IntSetType`, it can still use binary methods such as `intersect` since `IntSetType` is completely known in the context where the method gets evaluated.) Secondly, `intersect` still must have access to all of the methods of its parameter, and hence cannot take a parameter of type `#IntSetType2`. This means that the client can only intersect with sets that *haven't* been created with `newSet2`, (that is, whose type is completely known) which negates the whole purpose of the exercise. This second problem could be handled somewhat reasonably if the language provided a mechanism for testing and changing dynamic types by having the intersect method take a parameter of type `#IntSetType2` and then conditionally coerce it up to type `IntSetType`. Since all of this takes place within the method itself, no abstraction is lost. The first problem however

```
IntSetType2 = ObjectType
   add: proc(Integer);
   remove: proc(Integer);
   contains: func(Integer):Boolean;
   intersect: proc(MyType)
end;

newSet2 = function(): #IntSetType2
var
   newSet:IntSetType --as defined before
begin
   newSet := new(ListSetClass); --same ListSetClass
   return newSet;  -- IntSetType <# IntSetType2
end;
```

Figure 7.3: Abstract sets without modules

cannot in general be solved in this manner. While one could certainly coerce the type of an object from `#IntSetType2` to `IntSetType` and then send it the intersect message, this forces the code to rely explicitly on `IntSetType` being the actual implementation type of the object.

7.1.2 Friends

This conflict between the need for access to structural details and the need for abstraction that we see in in the set example is very common in object-oriented programming, making up a significant part of what is known as the *binary method* problem [BCC+96]. It is very frequently the case when designing object-oriented systems that methods must be built into objects to allow access to internal structure. (Note that if instance variables are publicly visible this is not a problem, but since such a system also eliminates all of the abstraction benefits of the object encapsulation, this is not a viable solution) This is very undesirable in terms of attempting to maintain abstraction and may also make it more difficult to maintain internal invariants since clients have potentially destructive access to internal structure.

C++ attempts to deal with this issue with a feature called *friends*. A C++ class may optionally include a list of functions and other classes that are to be allowed access to the class's private features. This allows the example above to be written relatively straightforwardly, since classes are defined to be friends with themselves [ES90]. Eiffel also provides a similar functionality by allowing the programmer to specify which features should

be exported to which classes [Mey92]. Other approaches to this problem include a concept called *multi-methods* [CL94] which allows dynamic message dispatch based on the dynamic type of the parameter to the method, and something called *friendly functions* [PT93] which provides similar functionality to C++'s friends in a type safe manner.

In $\mathcal{LOOM}$, we provide the essence of the friends idea without tampering with the semantics of classes, and class abstraction. Friends of a partially abstract type are simply functions or classes that occupy the same module as the type's implementation, and hence have access to the full interface of the type. By separating out the two levels of abstraction, we allow programmers to use the "friend" idea without having to worry about the complex changes in class behavior that can result. This is not the case in all such mechanisms. In particular, the C++ friend mechanism seems excessively complicated, with many obscure behaviors and restrictions that make it difficult to easily assess the affect of adding such functionality, particularly when combined with private/protected interfaces. Finally, note that in C++, all friends must be listed explicitly in the class declaration. As a result, new friends cannot be declared without modifying the original class. In $\mathcal{LOOM}$, new "friends" can be added at any point without modifying the original class. Of course, in the current implementation all code that wishes to use the friendly interface must reside in the same module as the class itself unless the friendly interface is exported, and hence adding a new friend requires modifying the module - not a significant improvement over C++.

7.1.3 Problems with the language

This last point brings up a definite shortcoming of the current language. Note that in figure 7.2, `SetOfInt` needs to import the class `IntOrdList` in order to inherit from it. This means that the complete type of `OrdListClassType` must be exported in the interface `IntOrdList`. Recall that one of the big advantages of modules is that they allow us to abstract away class implementations, eliminating inessential details and facilitating separate compilation as well as maintaining abstraction barriers. While we succeeded in doing this for the module `SetOfInt`, our design prevented us from doing so with `IntOrdList` since we were required to export `OrdListClass` (making the interface reflect the implementation). Moreover, by partially abstracting `IntSetType` and not exporting its class implementation, we have eliminated the possibility of ever inheriting from `ListSetClass` or of adding "friendly" functionality outside of the module in which it resides. Finally, note that in a real class library, we could very well want to implement `OrdListClass` as a subclass of a more general list class. If this was the case, we would want the interface `IntOrdList` to only partially reveal `OrdListType`, since the full type of `OrdList`objects would include methods for arbitrary changes to the list inherited from the parent list implementation that could not safely be exported to clients.

What really seems to be happening here is a regression of what might be called the peer/client problem from objects to modules. The original problem that we were attempt-

ing to address in example 7.1 is the need to provide certain classes (the peers) privileged implementation information and access, without violating the abstraction between the class and its clients. This was the original problem in figure 4.1 that we wished to address with the modules! While we have succeeded in eliminating this problem at one level, it seems to have reappeared again on the next level. While we are clearly better off than before, programmers still have to deal with the peer/client problem, and as result, must still compromise abstraction in order to provide needed functionality. In the next section, we discuss one possible solution to this problem.

7.2 Future Work

There are a number of interesting issues that remain to be examined in $\mathcal{LOOM}$ - both issues that we did not have time to address, and issues that in fact arose during the design or evaluation of the language. This year's work provides a solid, basic module system that implements what we felt was the essential functionality for programming in the large. However, it is very interesting to think about some of the more subtle and interesting additions to a module system that are possible.

7.2.1 Multiple interfaces/modules

In designing $\mathcal{LOOM}$, we struggled for a long time over the issue of how interfaces relate to implementations. There are essentially two issues that come up in trying to relate modules and interfaces - whether or not implementations are explicitly linked to interfaces, and if so, whether the correspondence between interfaces and implementations is one to one. So for instance, in a language like Eiffel which uses classes as the unit of modular abstraction, the interfaces are the types of the classes, which in Eiffel are in fact the classes themselves. As a result, there is no explicit control over the linking between interface and implementation. The interface (type) is bound to one and only one implementation (class).

At the other extreme, an **SML** structure can be associated with any number of different signatures, and any signature may be implemented by multiple structures. While **SML** requires that a structure definition be given a signature, the signature name chosen is not inherently part of the structure. So for example, a functor is defined as to take a parameter which has a signature as its "type", and in general, any structure whose internal implementation satisfies that signature may be passed in as a parameter. Clearly then, structures may be bound to different signatures. At the same time, note that there can be multiple structures using the same signature as their interface. **SML** uses the structure name in path names, as opposed to qualifying members with the signature (interface) name. This is a very different approach from that of most other languages, and while it is not something that we agree with whole-heartedly, we believe that there may be parts of this that are useful. For instance, is might seem reasonable to allow implementation modules

to deal with other implementation modules directly, particularly in the context of some sort of modular inherit mechanism. Moreover the manner in which **SML** allows multiple interfaces to a single structure raises interesting possibilities for implementing structures whose functionality is different in different contexts. This kind of construct is used frequently in object-oriented database programming, where it is common to want different objects to have different *views* of each other [Ala89].

Modula-3 supports a number of interesting features in this regard. To begin with, Modula-3 allows modules to implement multiple interfaces in a very straightforward way - modules simply specify more than one interface name in their export list. This is a very nice feature, and seems semantically clearer than the perhaps more powerful **SML** features. This particular functionality seems fairly important to a module system. In addition to a wide variety of programming styles that make it valuable to be able to specify separate interfaces to a common implementation, there is the very fundamental issue of the difference between the interfaces needed by client modules and those needed by peer modules. In general, client modules should be granted as little information as possible about details of the implementation. Indeed, in $\mathcal{LOOM}$, modules have proved to be very nice in situations where a good deal of complexity is necessary to support something with a relatively simple interface. Without a mechanism for hiding irrelevant (from the client perspective) details, the client must sort through unacceptable amounts of unwanted information to find the small bit that is valuable to them. Modules provide a mechanism for packaging these "irrelevant" details together and allowing clients to only deal with what is relevant to them. However, it is frequently the case that a peer module may wish to have access to more explicit implementation details in order to reuse the code. So for example in the set example from figure 7.2, module SetOfInt needs access to the actual implementation of ListOfInt in order to inherit from the ListClass. In general however, clients of ListOfInt may not want access to ListClass. More importantly, if they are granted access, then there is a loss of both abstraction and separate compilation. Since clients have access to implementation details, they may violate the abstraction barrier freely, and since client modules now depend on specific implementation details as opposed to simply a generic list interface, small implementation changes to the list may require otherwise unnecessary recompilation.

The solution to this, as Modula-3 handles it, is to allow clients to specify multiple interfaces. Client modules restrict themselves to using the abstract list interfaces, while peer modules that need access to implementation specific details use a more specific interface. This is a very nice solution to the problem, and one very well suited to the $\mathcal{LOOM}$ modules. Note that types exported through separate interfaces in $\mathcal{LOOM}$ could still potentially interact even if one or both were only partially revealed by using bounded genericity and hash types. While during the initial design process it was not clear that we necessarily wanted to support multiple interfaces, our experience in evaluating the language seems to suggest that this would provide a very useful functionality, and is certainly worth a close

examination in the future.

Modula-3 also supports an interesting mechanism whereby an interface may be implemented by multiple implementation modules. Multiple implementations may list the same interface, so long as each feature of the interface is implemented in exactly one of the modules. This allows for a very interesting variant to the *mix-in* idea supported by some object-oriented languages with multiple inheritance. Essentially, this allows the programmer to select parts of different modules and export them as a single unit. This seems like a fairly complicated mechanism to support, and it is not immediately clear how much functionality it adds. Nonetheless, it seems interesting enough to warrant a closer examination.

7.2.2 Parameterization over modules

Another issue that still needs some consideration is the possibility of allowing parameterization of modules over modules. $\mathcal{LOOM}$ already has very strong facilities for writing generic code, and it is not clear how much additional benefit could be received from parameterized modules, but it is certainly something to look into. We have already seen numerous references to **SML**'s approach to module parameterization, which results in behavior in some ways nicely coincident with more familiar facilities such as function calls and binding. In general however, it does not seem like an especially good idea to begin to treat implementations as values that can be accessed directly. While this allows some of **SML**'s more interesting behaviors, it also seems intrinsically linked with its inability to support separate compilation, and moreover makes it very difficult to avoid programming module dependencies into unrelated code simply because the implementation name must be used instead of that of a generic interface.

Modula-3 provides generics through parameterization over interfaces. Generic modules and interfaces can be written containing so-called *formal* import lists which specify one or more unused interface names imported by the module, but unbound to any specific interface. Generics are instantiated by binding *actual* interfaces to the formal interfaces. The semantics of this behavior is essentially equivalent to adding a list of imports renaming the actuals as the formals [Har92].

Parameterization over modules is a subject that needs to be evaluated very carefully. It seems in some senses to be a very powerful tool, but it is not clear that it is necessarily a useful or needed tool. The $\mathcal{LOOM}$ generic facilities provide much of the functionality that might otherwise require parameterized modules. However, the possibility of supporting at least a limited form of modular parameterization should certainly be kept open.

One interesting thing to note is that both of these concepts - multiple interfaces and modular parameterization - could potentially require that implementation modules be permitted to be bound to names. This was in fact the case in the original design for $\mathcal{LOOM}$, but was dropped for reasons of simplicity. Interestingly enough, this binding of names to modules seems to begin to reapproach the **SML** system at some level. Once implementa-

tions are given names, it seems reasonable to begin to be able to treat them as values, and questions about the distinction between interfaces and modules begin to reappear. Up until this point the $\mathcal{LOOM}$ module design has ended up paralleling choices made in Modula-3, despite perhaps more time spent studying the **SML**language. At this point, it seems like it may be worthwhile to examine the nature of the **SML** modules once again.

7.2.3 Storage allocation on modules?

The last issue that is worth looking more closely at is the possibility of supporting a more general notion of modules, including perhaps variables and executable code. In general, we have not found particularly convincing arguments for allowing blocks of executable code to be incorporated into modules to be executed at run-time. Modula-3 supports this as a means of initializing modules, but it is not clear how useful this is, and it is almost certainly useless without some way of allocating storage within a module.

There are on the other hand some quite reasonable, albeit not overwhelming arguments for allowing variables inside of modules. Essentially, this provides a functionality somewhere between that of an instance variable in an object and a static variable in a language like C or C++. This allows modules to keep state information relevant only to them within their own scope. While it is probably worth examining this possibility more closely at some point, there does not seem to be any significant shortcomings of the languages that would be addressed by this.

Chapter 8

Conclusion

8.1 Are modules and classes redundant?

There is still a question in many peoples minds about whether or not it is truly useful to supply both modules and classes as language features. As we mentioned briefly at the beginning of the chapter, languages such as SmallTalk and Eiffel use classes as the sole unit of modular organization and compilation. In some ways this is a nice solution as it provides a very simple interface for the user, and does not clutter the language with unnecessary features. Indeed, there does at times seem to be a certain amount of overlap in functionality between the two that makes it seem redundant to provide both, particularly in languages with very general models of one or both features. On closer examination however, we feel that there is a strong need for providing both, for a number of reasons.

The first issue that points to a need for both modules and classes is that there are functionalities to each that the other is unsuited to. We have seen this in a number of places. One of the most telling of these is the problems that object-oriented systems have with supporting separate compilation. As we noted before, languages that rely on class interfaces to provide the buffer between implementation details and clients suffer drastically in terms of separate compilation. Even in languages which do not allow or require instance variables to appear in the interface of the object/class (of which there are very few) it is almost impossible to keep implementation information from appearing in the types of methods. The real issue behind this is that classes are really too small. It is very common in writing programs that several entities will need to interact on a peer to peer basis. However in almost all cases it is virtually impossible and certainly undesirable to program an entire set of such entities into an object. Classes are designed to hold a single entity, focused around its data - they do not provide a good model for holding interacting entities. As a result of this, the interaction must take place outside of the object, and hence the object must provide explicit functionality to in essence open its scope to share structural

information. This requires that the types of structural information appear in the interface of the object. As a result of this, *any* non-trivial alteration to the implementation of the object will force all its clients to recompile, since they all are presented with an open interface.

This points up an immediate advantage of having *both* classes and modules - the fact that it allows to maintain three completely distinct levels of abstraction/scoping. Things may be private to an object, private to a module, or available to anyone who wants it. This allows much more subtle use of scoping and abstraction than any two levels of scoping. Also, note that with modules we say "available to anyone who *wants* it", not "everyone". This is also an important functionality that classes do not provide. In general, there is no way to use the class facility to specify client dependencies among classes (as opposed to inheritance dependencies). All classes are visible to all others in a pure object system. C++ gets around this by providing scoping within files, but this is essentially a primitive modular facility. The problem with this is that a compiler must be extremely smart if it is to avoid having to avoid recompile all classes whenever one changes its interface. With modules on the other hand, there is an explicit dependency list appended to the beginning of each module (the import list) which tells the compiler *exactly* which modules depend on which. This allows it to be much more selective in its recompilations.

Note that at the core there is a fundamental difference between classes and modules in that classes are run time entities, while modules are essentially compile time entities. This has two very noticeable effects. The first of these is actually an efficiency issue. Even the most efficient implementations of object-oriented systems must make sacrifices in terms of speed. The problem is that objects are run-time entities - indeed, a large part of the value of objects is their dynamic nature. Modules on the other hand are completely limited to compile time. While the compiler may have to do more work to patch together the system from many disparate pieces compiled at different times, once it does so it may hardwire in the exact location of everything from every module. Because modules do not need to be handled dynamically, there is never a need to do lookups in tables or indirections through other modules. This is certainly not an argument for eliminating objects - the overhead of lookup is more than worth the flexibility gained. It is however, a strong argument against trying to cram things that do not truly belong in classes into them simply to avoid having to use a separate language structure. This is ironically particularly prevalent in C++, where it is common to see code written using classes without any use of inheritance or subclassing at all, exactly as one might use a module.

The second issue to be considered with respect to the run-time/compile-time distinction is that of the difference between import and inheritance. Inheritance has three central functions. Firstly, it is a means of code reuse and organization. Inheritance allows code defined once in an object to be reused in a subclass without being rewritten. Secondly, inheritance is a means of changing the functionality of an operation associated with an object. By redefining a method, subclasses can specialize behavior for their own needs. Finally, inheritance is generally used as a means for either defining or reflecting the subtyping hierarchy.

All of these different uses require very complicated and sometimes restrictive rules. As a consequence of this, things like removing an inherited operation from a class's interface is generally not a good idea, even though it is frequently allowed. Eiffel claims that if this is done, it should not generate subtypes, but it doubtful that this is enforced, and C++ behaves similarly. In general, it is most common that all inherited features appear in the interface.

Import on the other hand primarily a method of specifying dependencies and gaining access in a scoped system. There is no idea of redefinition of imported elements, and perhaps most importantly, there is no need to worry about maintaining a safe subtype relation. It is frequently the case that modules will wish to import other modules for their private use without them appearing in the interface. Even in a transitive system this should be possible by importing to the implementation instead of the interface. It may also be the case that a module will export only parts of another module if this functionality is provided. All of this is quite acceptable with modules, but potentially very problematic with objects. It does not seem likely that a system that supports the kind of facilities desired for modular import will be able to enforce safe inheritance, while the rules for safe inheritance are liable to be too restrictive for import.

Finally, recall that classes really do not provide the kind of abstraction we need for true modular programming. Classes must almost always export too much information, and hence if we are using the class as our fundamental unit of abstraction, then we must rely on clients not to take advantage of privileged methods, either maliciously or accidentally. Moreover, class systems do not generally provide any support for partial abstraction, or indeed any type abstraction other than the ability in some systems to abstract away private class features. In general the abstraction facilities of a powerful module system are almost impossible to duplicate with classes, and it is not clear that there is any nice way to extend classes to provide similar functionality.

These issues lead us to feel that it is worth having both modules and objects in the same language, and that indeed it is beneficial to do so. Without modules, languages with classes are forced to try and add functionality to classes to simulate modular features, and vice versa. Rather than complicating things by having both modules and classes, it is possible to actually simplify things, since each component's required functionality gets smaller and more well-defined. We feel that a strong module system with good support for type abstraction, separate compilation and namespace management is an essential feature of any language.

Bibliography

[AC95] Martin Abadi and Luca Cardelli. On subtyping and matching. In *Proceedings ECOOP '95*, pages 145–167, 1995.

[Ala89] Suad Alagić. *Object-Oriented Database Programming.* Texts and Monographs in Computer Science. Springer-Verlag New York Inc., 1989.

[BCC+96] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Objects Group, Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object-Oriented Systems*, 1996. to appear.

[BCD+93] K. Bruce, J. Crabtree, A. Dimock, R. Muller, T. Murtagh, and R. van Gent. Safe and decidable type checking in an object-oriented language. In *Proc. ACM Symp. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 29–46, 1993.

[BCK94] K. Bruce, J. Crabtree, and G. Kanapathy. An operational semantics for TOOPLE: A statically-typed object-oriented programming language. In S. Brookes, M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Semantics*, pages 603–626. LNCS 802, Springer-Verlag, 1994.

[Bru93] K. Bruce. Safe type checking in a statically typed object-oriented programming language. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 285–298, 1993.

[Bru94] K. Bruce. A paradigmatic object-oriented programming language: design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206, 1994. An earlier version of this paper appeared in the 1993 POPL Proceedings.

[BSvG95] Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language, extended abstract. In *ECOOP '95*, pages 27–51. LNCS 952, Springer-Verlag, 1995. A complete version of this paper with full proofs is available via http://www.cs.williams.edu/~kim/.

[BvG93] Kim B. Bruce and Robert van Gent. TOIL: A new type-safe object-oriented imperative language. Technical report, Williams College, 1993.

[Car89] L. Cardelli. Typeful programming. Technical Report 45, DEC Systems Research Center, 1989. Presented at IFIP Advanced Seminar on Formal Descriptions of Programming Concepts.

[CL94] Craig Chambers and Gary T. Leavens. Typechecking and modules for multi-methods. In *OOPSLA Proceedings*, 1994. to appear.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[DGLM95] Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Meyers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *Proc. ACM Symp. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 156–168, 1995.

[ES90] Margaret A. Ellis and Bjarne Stroustrop. *The annotated C^{++} reference manual.* Addison-Wesley, 1990.

[Har92] Samuel P. Harbison. *Modula-3.* Prentice-Hall, Inc., New Jersey, 1992.

[HMM86] R. Harper, D.B. MacQueen, and R. Milner. Standard ML. Technical Report ECS–LFCS–86–2, Lab. for Foundations of Computer Science, University of Edinburgh, March 1986.

[Joi95] Joint Technical Committee ISO/IEC JTC 1. *Ada 95 Reference Manual.* Intermetrics, Inc., 1995.

[Jon96] Mark P. Jones. Using parameterized signatures to express modular structure. In *23rd ACM Symp. Principles of Programming Languages*, pages 68–78, 1996.

[KLM94] Dinesh Katiyar, David Luckham, and John Mitchell. A type system for prototyping languages. In *21st ACM Symp. Principles of Programming Languages*, pages 138–150, 1994.

[KM89] P.C. Kanellakis and J.C. Mitchell. Polymorphic unification and ML typing. In *16th ACM Symposium on Principles of Programming Languages*, pages 105–115, 1989.

[Ler94] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proc. 21st ACM Symp. on Principles of Programming Languages*, pages 109–122, 1994.

[Ler95] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Proc. 22nd ACM Symp. on Principles of Programming Languages*, 1995.

[Lis88] Barbara Liskov. Data abstraction and hierarchy. In *OOPSLA '87 Addendum to the Proceedings*, pages 17–34. ACM SIGPLAN Notices,23(5), May 1988.

[LSAS77] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in clu. *Comm. ACM*, 20:564–576, 1977.

[Mac85] D.B. MacQueen. Modules for Standard ML. *Polymorphism*, 2(2), 1985. 35 pages. An earlier version appeared in Proc. 1984 ACM Symp. on Lisp and Functional Programming.

[Mey92] B. Meyer. *Eiffel: the language*. Prentice-Hall, 1992.

[Nel91] Greg Nelson, editor. *Systems Programming in Modula-3*. Prentice-Hall, Inc., New Jersey, 1991.

[Pie92] Benjamin C. Pierce. Bounded quantification is undecidable. In *Proc 19th ACM Symp. Principles of Programming Languages*, pages 305–315, 1992.

[PT93] Benjamin C. Pierce and David N. Turner. Statically typed friendly functions via partially abstract types. Technical Report ECS-LFCS-93-256, University of Edinburgh, 1993.

[Sch94] Angela Schuett. *Parametric Polymorphism in a type-safe, object-oriented programming language*. Williams College Senior Honors Thesis, 1994.

[Tes85] L. Tesler. Object Pascal report. Technical Report 1, Apple Computer, 1985.

[vG93] Robert van Gent. *TOIL: An imperative type-safe object-oriented language*. Williams College Senior Honors Thesis, 1993.

Appendix A

Complete grammar for *LOOM*

A.1 Module Syntax

Module ::= *Interface* | *Implementation* | *Program*

Interface ::= `Interface` <id> *ImportList DeclList* `end`

Implementation ::= `Module Implements` <id> *ImportList*;
`Type` *AbbrevList* `Const` *ConstList* `end`

Program ::= `Program` <id> *ImportList AbbrevList UnitBlock*

ImportList ::= `Imports` *IdList*; | ∅

DeclList ::= *Revelation*; *DeclList* | ∅

Revelation ::= *PartialRev* | *Assertion*

PartialRev ::= <id> <# *TypeExp*

Assertion ::= <id> = *TypeExp* |
::= <id>: *TypeExp*

$$ConstList \ ::= \ \text{<id>} \ = \ Expr\text{: } TypeExp$$

$$IdList \ ::= \ \text{<id>}; IdList \ | \ \text{<id>}$$

$$PathName \ ::= \ < id > \ | \ < interfacename > :: < id >$$

A.2 Base Syntax

```
Program =             PROGRAM id ";" InputFile Import VarIdlist AbbrevList
                          UnitBlock "."

InputFile =           [INPUTFILE "=" STRNG [";"]]

AbbrevList =          [Type AbbrevDeclList ]

AbbrevDeclList =      id "=" TypeExp {";" id "=" TypeExp}
                          [";"]

VarIDList =           id {"," id}

IDList  =             [VarIDList]

UnitBlock =           ConstList VarList BEGIN StmtList END

FuncBlock  =          ":" TypeExp ConstList VarList BEGIN
StmtList                  RETURN Exp [";"] END

ConstList  =          [CONST InitializedVarDeclList]

InitializedVarDeclList = id "=" FuncExp [";"]
                       | id "=" Exp ":" TypeExp [";"]
                       | id "=" FuncExp ";" InitializedVarDeclList
                       | id "=" Exp ":" TypeExp ";"  InitializedVarDeclList

VarList  =              [VAR VarDeclList]

VarDeclList =           (VarIDList ":" TypeExp)
```

```
                        {";" VarIDList ":" TypeExp} [";"]

TFuncParamDeclList =   VarIDList  "<#" TypeExp [";"]
                          | VarIDList "<#" TypeExp ";"
                              TFuncParamDeclList
FuncParamDeclList =    VarIDList  "<#" TypeExp [";"]
                          | VarIDList ":" TypeExp [";"]
                          | VarIDList  "<#" TypeExp ";"
                              FuncParamDeclList
                          | VarIDList ":" TypeExp ";"
                              FuncParamDeclList

InitializedVarList =   [VAR InitializedVarDeclList]

MethodList =           [METHODS MethodSection]

MethodSection =        (HIDDEN Methods | VISIBLE Methods)
                           {HIDDEN Methods | VISIBLE Methods}

Methods =              id "=" FuncExp {";" id "=" FuncExp}[;]

NonEmptyTypeList  =    TypeExp {"," TypeExp}

TypeList =             [NonEmptyTypeList]

MethodTypes =          [METHODS MethodTypeSection]

MethodTypeSection =    (HIDDEN VarDeclList | VISIBLE
                          VarDeclList)
                          {HIDDEN VarDeclList | VISIBLE
                          VarDeclList}

ObjectMethods =        [[METHODS][VarDeclList]]


TypeExp =                id
                       | id "["NonEmptyTypeList"]"
                       | id "<#" TypeExp
                       | MYTYPE
                       | BOOL
```

```
                    | INTEGER
                    | REAL
                    | STRING
                    | TOPOBJTYPE
                    | TOPCLASSTYPE
                    | FUNC "(" TypeList ")" ":" TypeExp
                    | PROC "(" TypeList ")"
                    | CLASSTYPE [INCLUDE TypeExp[MODIFYING
                        IDList ";"]] VarList MethodTypes
                        END
                    | OBJECTTYPE [INCLUDE TypeExp
                        [MODIFYING IDList
                        ";"]ObjectMethods]
                        END
                    | TFUNC "[" TFuncParamDeclList "]"
                        TypeExp
                    | "(" id ")"
                    | "#" TypeExp


StmtList =          {Stmt}

Stmt =                Exp ":=" Exp ";"
                    | WHILE Exp DO StmtList END ";"
                    | IF Exp THEN StmtList {ELSIF Exp THEN
                        StmtList} [ELSE StmtList] END ";"
                    | Exp "(" ParamList ")" ";"
                    | FOR Exp ":=" Exp TO Exp [BY Exp] DO
                        StmtList
                        END ";"

FuncExp =           FUNCTION "(" [FuncParamDeclList] ")"
                        FuncBlock
                    | PROCEDURE "("[FuncParamDeclList] ")"
                        UnitBlock

ParamList =         {Param}

Param =               Exp
                    | TypeExp
```

```
Exp =                    id
                       | integer
                       | real
                       | boolean
                       | string
                       | SELF
                       | NIL
                       | TOP
                       | Exp "(" ParamList ")"
                       | "(" Exp INFIXOP Exp ")"
                       | Exp "." id
                       | SUPER "." id
                       | FuncExp
                       | NEW "(" Exp ")"
                       | CLASS [INHERIT Exp [MODIFYING
                           IDList] ";"]InitializedVarList
                           MethodList END
                       | "(" id ")"
```

Appendix B

Complete type checking rules for *LOOM*

Definition B.0.1 (Type Constraint System) *Relations of the form $\sigma <\# \tau$ where σ and τ are type expressions, are said to be* type constraints. *A* type constraint system *is defined as follows:*

1. *The empty set, ϵ, is a type constraint system. In general we will denote an empty type constraint system as $\mathcal{C}_o$.*
2. *If $\mathcal{C}$ is a type constraint system, τ is a type variable or a type of the form $ObjectType(Mytype)\sigma$, and $t <\# \tau$ is a type constraint such that the type variable t does not appear free in $\mathcal{C}$ or τ, then $C \cup \{t <\# \tau\}$ is a type constraint system.*

So, C keeps track of matching constraints placed on type variables in the current scope. E holds variable names and the types associated with them. The formal definition of E is as follows:

Definition B.0.2 (Type Assignment) *A type assignment $\mathbf{E}$ is a finite set of associations between variables and type expressions of the form $x{:}\,\tau$, where each x is unique in $\mathbf{E}$. If the relation $x{:}\,\tau \in \mathbf{E}$, then we write $\mathbf{E}(x) = \tau$.*

Definition B.0.3 (Equivalence Table) *Relations of the form $t = \sigma$ are* type equivalences. *An* equivalence table *is defined as such:*

1. *The empty set, ϵ, is an equivalence table. In general we will denote an empty equivalence table as ST_o.*
2. *If ST is an equivalence table, τ is a type, t is a type variable that does not occur in ST, and $t = \tau$ is a type equivalence, then $\mathsf{ST} \cup \{t = \tau\}$ is an equivalence table.*

B.1 Matching rules

We say that $\{m_i{:}\,\tau_i\}^{1\leq i\leq n}$ *extends* $\{m_i{:}\,\tau_i\}^{1\leq i\leq k}$, if $n \geq k$.

$Var(<\#)$ $$C \cup \{t <\# \tau\} \vdash t <\# \tau,$$

$Refl(<\#)$ $$\frac{C \vdash \tau <\# \top}{C \vdash \tau <\# \tau,}$$

$Trans(<\#)$ $$\frac{C \vdash \sigma <\# \tau}{C \cup \{t <\# \sigma\} \vdash t <\# \tau}\ ,$$

$VisObjType(<\#)$ $$\frac{\sigma \ \mathit{extends}\ \sigma',\quad \tau \ \mathit{extends}\ \tau'}{C \vdash \mathbf{VisObjType}(\sigma,\tau) <\# \mathbf{VisObjType}(\sigma',\tau')}\ ,$$

$ObjectType(<\#)$ $$\frac{\tau \ \mathit{extends}\ \tau'}{C \vdash \mathit{ObjectType}\ \tau <\# \mathit{ObjectType}\ \tau'}\ ,$$

B.2 Base language type checking rules

Type Assignment Rules (Program):

$$\frac{\mathcal{C}_0, \mathbf{E}_0 \vdash \mathit{Block}\text{: COMMAND}}{\mathcal{C}_0, \mathbf{E}_0 \vdash \text{program } p; \mathit{Block}.\ :\ \text{PROGRAM}}$$

Type Assignment Rules (Declarations):
Note that these type assignment rules for declarations actually produce a new environment, $\mathbf{E}$.

$ConstDecls$ $$\frac{\mathcal{C}, \mathbf{E} \vdash \mathit{CDcLst} \rhd \mathbf{E}'}{\mathcal{C}, \mathbf{E} \vdash \text{const } \mathit{CDclLst} \rhd \mathbf{E}'}$$

$ConstDcl+$ $$\frac{\mathcal{C}, \mathbf{E} \vdash \mathit{CDcl} \rhd \mathbf{E}',\quad \mathcal{C}, \mathbf{E}' \vdash \mathit{CDclLst} \rhd \mathbf{E}''}{\mathcal{C}, \mathbf{E} \vdash \mathit{CDcl};\ \mathit{CDclLst} \rhd \mathbf{E}''}$$

where x does not occur in $\mathbf{E}$.

ConstDcl
$$\frac{\mathcal{C}, \mathbf{E} \vdash M\colon \tau}{\mathcal{C}, \mathbf{E} \vdash x = M\colon \tau \rhd \mathbf{E} \cup \{x\colon \tau\}}$$

where x does not occur in $\mathbf{E}$.

VarDcls
$$\frac{\mathcal{C}, \mathbf{E} \vdash \mathit{VDcLst} \rhd \mathbf{E}'}{\mathcal{C}, \mathbf{E} \vdash \text{var } \mathit{VDclLst} \rhd \mathbf{E}'}$$

VarDcl+
$$\frac{\mathcal{C}, \mathbf{E} \vdash \mathit{VDcl} \rhd \mathbf{E}', \quad \mathcal{C}, \mathbf{E}' \vdash \mathit{VDclLst} \rhd \mathbf{E}''}{\mathcal{C}, \mathbf{E} \vdash \mathit{VDcl}; \mathit{VDclLst} \rhd \mathbf{E}''}$$

VarDcl
$$\mathcal{C}, \mathbf{E} \vdash x\colon \tau \rhd \mathbf{E} \cup \{x\colon \mathit{ref}\ \tau\}$$

where x does not occur in $\mathbf{E}$.

Type Assignment Rules (Blocks):

Block
$$\frac{\begin{array}{c}\mathcal{C}, \mathbf{E} \vdash \mathit{CDcls} \rhd \mathbf{E}_1, \quad \mathcal{C}, \mathbf{E}_1 \vdash \mathit{VDcls} \rhd \mathbf{E}_2, \\ \mathcal{C}, \mathbf{E}_2 \vdash S\colon \text{COMMAND}, \quad \mathcal{C}, \mathbf{E}_2 \vdash M\colon \sigma\end{array}}{\mathcal{C}, \mathbf{E} \vdash \mathit{CDcls}\ \mathit{VDcls} \text{ begin } S \text{ return } M \text{ end}\colon \sigma}$$

Type Assignment Rules (Commands):

Assn
$$\frac{\mathcal{C}, \mathbf{E} \vdash x\colon \mathit{ref}\ \tau, \quad \mathcal{C}, \mathbf{E} \vdash M\colon \tau}{\mathcal{C}, \mathbf{E} \vdash x\colon = M\colon \text{COMMAND}}$$

Cond
$$\frac{\mathcal{C}, \mathbf{E} \vdash B\colon \mathit{Bool}, \ \mathcal{C}, \mathbf{E} \vdash S\colon \text{COMMAND}, \quad \mathcal{C}, \mathbf{E} \vdash T\colon \text{COMMAND}}{\mathcal{C}, \mathbf{E} \vdash \text{if } B \text{ then } S \text{ else } T \text{ end}\colon \text{COMMAND}}$$

While
$$\frac{\mathcal{C}, E \vdash B\colon \mathit{Bool}, \ \mathcal{C}, \mathbf{E} \vdash S\colon \text{COMMAND}}{\mathcal{C}, \mathbf{E} \vdash \text{while } B \text{ do } S \text{ end}\colon \text{COMMAND}}$$

StmtList
$$\frac{\mathcal{C}, \mathbf{E} \vdash S\colon \text{COMMAND}, \ \mathcal{C}, \mathbf{E} \vdash T\colon \text{COMMAND}}{\mathcal{C}, \mathbf{E} \vdash S;\ T\colon \text{COMMAND}}$$

Type Assignment Rules (Expressions):

Program $\quad \mathcal{C}, \mathbf{E} \vdash \text{OK: PROGRAM}$

Command $\quad \mathcal{C}, \mathbf{E} \vdash \text{command: COMMAND}$

Nil $\quad \mathcal{C}, \mathbf{E} \vdash \text{nil}: \bot$

Constant $\quad \mathcal{C}, \mathbf{E} \vdash c^\tau : \tau, \quad \text{for } c^\tau \in \mathcal{C}$

Var $\quad \mathcal{C}, \mathbf{E} \vdash x : \tau, \quad \text{if } \mathbf{E}(x) = \tau$

Value
$$\frac{\mathcal{C}, \mathbf{E} \vdash M : \mathit{ref}\ \tau}{\mathcal{C}, \mathbf{E} \vdash \text{val } M : \tau}$$

Function
$$\frac{\mathcal{C}, \mathbf{E} \cup \{v : \sigma\} \vdash \mathit{Block} : \tau}{\mathcal{C}, \mathbf{E} \vdash \text{function } (v : \sigma)\ \mathit{Block}:\ (\mathit{Func}\ (\sigma) : \tau)}$$

where σ may be of the form $\#\gamma$.

BdPolyFunc
$$\frac{\mathcal{C} \cup \{t <\!\# \gamma\}, \mathbf{E}\ \vdash \mathit{Block} : \tau}{\mathcal{C}, \mathbf{E} \vdash \text{function } (t <\!\# \gamma)\ \mathit{Block}:\ (\mathit{Func}\ (t <\!\# \gamma) : \tau)}$$

FuncAppl
$$\frac{\mathcal{C}, \mathbf{E} \vdash f : \mathit{Func}(\sigma) : \tau \quad \mathcal{C}, \mathbf{E} \vdash M : \sigma}{\mathcal{C}, \mathbf{E} \vdash f(M) : \tau}$$

BdPolyFuncAppl
$$\frac{\mathcal{C}, \mathbf{E} \vdash f : \mathit{Func}(t <\!\# \gamma) : \tau \quad \mathcal{C} \vdash \sigma <\!\# \gamma}{\mathcal{C}, \mathbf{E} \vdash f[\sigma] : \tau[\sigma/t]}$$

Class
$$\frac{\mathcal{C}^{IV}, \mathbf{E} \vdash a : \sigma, \quad \mathcal{C}^{METH}, \mathbf{E}^{METH} \vdash e : \tau}{\mathcal{C}, \mathbf{E} \vdash \text{class}(a, e) : \mathbf{ClassType}(\sigma, \tau)}$$

where $\mathcal{C}^{IV} = \mathcal{C} \cup \{\mathbf{MyType} <\# \mathit{ObjectType}\ \tau\}$,
$\mathcal{C}^{METH} = \mathcal{C}^{IV} \cup \{\mathbf{SelfType} <\# \mathbf{VisObjType}(\sigma, \tau)\}$,
$\mathbf{E}^{METH} = \mathbf{E} \cup \{\mathit{self}\colon \mathbf{SelfType}, \mathit{close}\colon \mathit{Func}(\mathbf{SelfType})\colon \mathbf{MyType}\}$

Neither **MyType** nor **SelfType** may occur free in $\mathcal{C}$ or $\mathbf{E}$.

σ and τ must both be record types, while the components of τ must be function types.

Inherits

$$\frac{\begin{array}{c}\mathcal{C}, \mathbf{E} \vdash c\colon \mathbf{ClassType}(\{v_1\colon \sigma_1; \ldots; v_m\colon \sigma_m\}, \{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}),\\ \mathcal{C}^{IV}, \mathbf{E} \vdash a_{m+1}\colon \sigma_{m+1}, \quad \mathcal{C}^{IV}, \mathbf{E} \vdash a'_1\colon \sigma_1,\\ \mathcal{C}^{METH}, \mathbf{E}^{METH} \vdash e_{n+1}\colon \tau_{n+1}, \quad \mathcal{C}^{METH}, \mathbf{E}^{METH} \vdash e'_1\colon \tau_1\end{array}}{\begin{array}{l}\mathcal{C}, \mathbf{E} \vdash \text{class inherit } c \text{ modifying } v_1, m_1;\\ \qquad (\{v_1 = a'_1\colon \sigma_1, v_{m+1} = a_{m+1}\colon \sigma_{m+1}\}, \{m_1 = e'_1\colon \tau_1, m_{n+1} = e_{n+1}\colon \tau_{n+1}\})\colon\\ \qquad\quad \mathbf{ClassType}(\{v_1\colon \sigma_1; \ldots; v_{m+1}\colon \sigma_{m+1}\}, \{m_1\colon \tau_1; m_2\colon \tau_2; \ldots; m_{n+1}\colon \tau_{n+1}\})\end{array}}$$

where $\mathcal{C}^{IV} = \mathcal{C} \cup \{\mathbf{MyType} <\# \mathit{ObjectType}\ \{m_1\colon \tau_1; \ldots; m_{n+1}\colon \tau_{n+1}\}\}$,
$\mathcal{C}^{METH} = \mathcal{C}^{IV} \cup \{\mathbf{SelfType} <\# \mathbf{VisObjType}(\{v_1\colon \sigma_1; \ldots; v_{m+1}\colon \sigma_{m+1}\}, \{m_1\colon \tau_1; \ldots; m_{n+1}\colon \tau_{n+1}\})\}$
$\mathbf{E}^{METH} = \mathbf{E} \cup \{\mathit{self}\colon \mathbf{SelfType}, \mathit{close}\colon \mathit{Func}(\mathbf{SelfType})\colon \mathbf{MyType}, \mathit{super}\colon \mathbf{SelfType} \to \{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}\}$

Neither **MyType** nor **SelfType** may occur free in $\mathcal{C}$ or $\mathbf{E}$.

Object
$$\frac{\mathcal{C}, \mathbf{E} \vdash c\colon \mathbf{ClassType}(\sigma, \tau)}{\mathcal{C}, \mathbf{E} \vdash \text{new } c\colon \mathit{ObjectType}\ \tau}$$

Weakening
$$\frac{\mathcal{C}, \mathbf{E} \vdash e\colon \tau}{\mathcal{C}, \mathbf{E} \vdash e\colon \#\tau}$$

Subsump
$$\frac{\mathcal{C} \vdash \sigma <\# \tau, \quad \mathcal{C}, \mathbf{E} \vdash e\colon \#\sigma}{\mathcal{C}, \mathbf{E} \vdash e\colon \#\tau}$$

Msg
$$\frac{\mathcal{C}, \mathbf{E} \vdash o\colon \gamma, \quad \mathcal{C} \vdash \gamma <\# \mathit{ObjectType}\{m\colon \tau\}}{\mathcal{C}, \mathbf{E} \vdash o \Leftarrow m\colon \tau[\gamma/\mathbf{MyType}]}$$

Msg#
$$\frac{\mathcal{C}, \mathbf{E} \vdash o\colon \#\mathit{ObjectType}\{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}}{\mathcal{C}, \mathbf{E} \vdash o \Leftarrow m_i\colon \tau_i[\#\mathit{ObjectType}\{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}/\mathbf{MyType}]}$$

Only if all occurrences of **MyType** in τ are positive.

InstVar
$$\frac{\mathcal{C}, \mathbf{E} \vdash o\colon \#\mathbf{VisObjType}(\{v_1\colon \sigma_1; \ldots; v_n\colon \sigma_n\}, \tau)}{\mathcal{C}, \mathbf{E} \vdash o.v_i\colon \mathit{ref}(\sigma_i)}$$

B.3 Module type checking rules

Lists of declarations. (Reveals relation)

$$(DecL{:}\emptyset) \qquad \mathcal{S}, \mathcal{M}_B^e, \mathcal{M}_B^i \vdash emptyDecLst \overset{\mathbf{m}}{\rightsquigarrow} (\mathcal{M}_B^e, \mathcal{M}_B^i)$$

$$(DecL{:}<\#) \qquad \frac{\mathcal{S} \cup \{t\}, \mathcal{M}_B^{e\,\prime}, \mathcal{M}_B^{i\,\prime} \vdash DecLst \overset{\mathbf{m}}{\rightsquigarrow} (\mathcal{M}_B^{e\,\prime\prime}, \mathcal{M}_B^{i\,\prime\prime})}{\mathcal{S}, \mathcal{M}_B^e, \mathcal{M}_B^i \vdash t <\# \tau;\ DecLst \overset{\mathbf{m}}{\rightsquigarrow} (\mathcal{M}_B^{e\,\prime\prime}, \mathcal{M}_B^{i\,\prime\prime})}$$

where $\mathcal{M}_B^{e\,\prime} = \mathcal{M}_B^e \overset{\mathcal{C}}{\hookleftarrow} (\text{B::t} <\# \mathsf{R_B}(\tau, \mathcal{S}))$ and $\mathcal{M}_B^{i\,\prime} = \mathcal{M}_B^i \overset{\mathcal{C}}{\hookleftarrow} (t <\# \tau)$

$$(DecL{:}=) \qquad \frac{\mathcal{S} \cup \{t\}, \mathcal{M}_B^{e\,\prime}, \mathcal{M}_B^{i\,\prime} \vdash DecLst \overset{\mathbf{m}}{\rightsquigarrow} (\mathcal{M}_B^{e\,\prime\prime}, \mathcal{M}_B^{i\,\prime\prime})}{\mathcal{S}, \mathcal{M}_B^e, \mathcal{M}_B^i \vdash t = \tau;\ DecLst \overset{\mathbf{m}}{\rightsquigarrow} (\mathcal{M}_B^{e\,\prime\prime}, \mathcal{M}_B^{i\,\prime\prime})}$$

where $\mathcal{M}_B^{e\,\prime} = \mathcal{M}_B^e \overset{\mathsf{ST}}{\hookleftarrow} (\text{B::t} = \mathsf{R_B}(\tau, \mathcal{S}))$ and $\mathcal{M}_B^{i\,\prime} = \mathcal{M}_B^i \overset{\mathsf{ST}}{\hookleftarrow} (t = \tau)$

$$(DecL{:}const) \qquad \frac{\mathcal{S}, \mathcal{M}_B^{e\,\prime}, \mathcal{M}_B^{i\,\prime} \vdash DecLst \overset{\mathbf{m}}{\rightsquigarrow} (\mathcal{M}_B^{e\,\prime\prime}, \mathcal{M}_B^{i\,\prime\prime})}{\mathcal{S}, \mathcal{M}_B^e, \mathcal{M}_B^i \vdash i{:}\,\tau;\ DecLst \overset{\mathbf{m}}{\rightsquigarrow} (\mathcal{M}_B^{e\,\prime\prime}, \mathcal{M}_B^{i\,\prime\prime})}$$

where $\mathcal{M}_B^{e\,\prime} = \mathcal{M}_B^e \overset{\mathbf{E}}{\hookleftarrow} (\text{B::e:}\, \mathsf{R_B}(\tau, \mathcal{S}))$ and $\mathcal{M}_B^{i\,\prime} = \mathcal{M}_B^i \overset{\mathbf{E}}{\hookleftarrow} (e{:}\,\tau)$

Definition Lists

$$(DefL{:}\emptyset) \qquad \mathcal{C}, \mathbf{E}, \mathsf{ST} \vdash emptyDefLst$$

$$(DefL{:}=) \qquad \frac{\mathcal{C}, \mathbf{E}, \mathsf{ST} \cup \{t = \tau\} \vdash DefLst}{\mathcal{C}, \mathbf{E}, \mathsf{ST} \vdash t = \tau; DefLst}$$

$$(DefL{:}const) \qquad \frac{\mathcal{C},\mathbf{E},\mathsf{ST} \vdash e{:}\tau' \quad \mathcal{C},\mathsf{ST} \vdash \tau = \tau' \quad \mathcal{C},\mathbf{E} \cup \{f{:}\tau\},\mathsf{ST} \vdash DefLst}{\mathcal{C},\mathbf{E},\mathsf{ST} \vdash f = e{:}\tau; DefLst}$$

Consistent with relation

$$(\lessdot\!*{:}\emptyset) \qquad \mathcal{C},\mathsf{ST} \vdash emptyDefLst \lessdot\!* (\{\},\{\},\mathsf{ST}_A)$$

$$(\lessdot\!*{:}\lessdot\!\#) \qquad \frac{(t \lessdot\!\# \tau') \in \mathcal{C}_A, \quad \mathcal{C},\mathsf{ST} \vdash \tau \lessdot\!\# \tau' \quad \mathcal{C},\mathsf{ST} \cup \{t = \tau\} \vdash rest \lessdot\!* (\mathcal{C}_A - \{t \lessdot\!\# \tau'\},\mathbf{E}_A,\mathsf{ST}_A)}{\mathcal{C},\mathsf{ST} \vdash t = \tau; rest \lessdot\!* (\mathcal{C}_A,\mathbf{E}_A,\mathsf{ST}_A)}$$

$$(\lessdot\!*{:}=) \qquad \frac{(t = \tau') \in \mathsf{ST}_A, \quad \mathcal{C},\mathsf{ST} \vdash \tau = \tau' \quad \mathcal{C},\mathsf{ST} \cup \{t = \tau\} \vdash rest \lessdot\!* (\mathcal{C}_A,\mathbf{E}_A,\mathsf{ST}_A - \{t = \tau'\})}{\mathcal{C},\mathsf{ST} \vdash t = \tau; rest \lessdot\!* (\mathcal{C}_A,\mathbf{E}_A,\mathsf{ST}_A)}$$

$$(\lessdot\!*{:}hid) \qquad \frac{(t \lessdot\!\# \tau') \notin \mathcal{C}_A, \quad (t = \tau') \notin \mathsf{ST}_A \quad \mathcal{C},\mathsf{ST} \cup \{t = \tau\} \vdash rest \lessdot\!* (\mathcal{C}_A,\mathbf{E}_A,\mathsf{ST}_A)}{\mathcal{C},\mathsf{ST} \vdash t = \tau; rest \lessdot\!* (\mathcal{C}_A,\mathbf{E}_A,\mathsf{ST}_A)}$$

$$(\lessdot\!*{:}const) \qquad \frac{(f{:}\tau') \in \mathbf{E}_A, \quad \mathcal{C},\mathsf{ST} \vdash \tau = \tau' \quad \mathcal{C},\mathsf{ST} \vdash rest \lessdot\!* (\mathcal{C}_A,\mathbf{E}_A - \{f{:}\tau'\},\mathsf{ST}_A)}{\mathcal{C},\mathsf{ST} \vdash f = e{:}\tau; rest \lessdot\!* (\mathcal{C}_A,\mathbf{E}_A,\mathsf{ST}_A)}$$

$$(<*: hid\ const) \qquad \frac{(f{:}\tau') \notin \mathbf{E}_A \quad \mathcal{C}, \mathsf{ST} \vdash rest\ <*\ (\mathcal{C}_A, \mathbf{E}_A, \mathsf{ST}_A)}{\mathcal{C}, \mathsf{ST} \vdash f = e{:}\tau; rest\ <*\ (\mathcal{C}_A, \mathbf{E}_A, \mathsf{ST}_A)}$$

Module Lists

$$ModLst{:}\emptyset \qquad \mathcal{I}, \mathbf{E} \vdash emptyModLst{:}\mathcal{I}$$

$$ModLst{:}Program \qquad \frac{abbrevs \Rightarrow \mathsf{ST}_P \quad \mathcal{I}(B) = ((\mathcal{C}_B, \mathbf{E}_B, \mathsf{ST}_B), \mathcal{M}^i_B) \quad \mathcal{C}_B, \mathbf{E}_B, \mathsf{ST}_B \cup \mathsf{ST}_P \vdash Block\mathcal{I}, \mathbf{E} \vdash ModLst{:}\mathcal{I}'}{\mathcal{I}, \mathbf{E} \vdash Program\ P\ Import\ B\ Abrvs\ Block; ModLst{:}Program}$$

$$ModLst{:}Int \qquad \frac{\mathcal{S}_o, \mathcal{M}_o, \mathcal{M}^e_B \vdash DecLst \overset{\mathbf{m}}{\rightsquigarrow} (\mathcal{M}^e_A, \mathcal{M}^i_A) \quad \mathcal{I} \cup \{(A, \mathcal{M}^e_A, \mathcal{M}^i_A)\}, \mathbf{E} \vdash ModLst{:}\mathcal{I}'}{\mathcal{I}, \mathbf{E} \vdash Interface\ A\ Import\ B\ DecLst;\ ModLst{:}\mathcal{I}'}$$

where $\mathcal{I}(B) = (\mathcal{M}^e_B, \mathcal{M}^i_B)$

$$(ModLst{:}Imp) \qquad \frac{\mathcal{I}(A) = (\mathcal{M}^e_A, (\mathcal{C}_A, \mathbf{E}_A, \mathsf{ST}_A)) \quad \mathcal{I}(B) = ((\mathcal{C}_B, \mathbf{E}_B, \mathsf{ST}_B), \mathcal{M}^i_B) \quad \mathsf{ST}_B, \mathcal{C}_B \vdash DefList\ <*\ (\mathcal{C}_A, \mathbf{E}_A, \mathsf{ST}_A) \quad \mathcal{C}_B, \mathbf{E}_B, \mathsf{ST}_B \cup \mathsf{ST}_A \vdash DefLst,\ \ \mathcal{I}, \mathbf{E} \vdash rest{:}\mathcal{I}'}{\mathcal{I}, \mathbf{E} \vdash Module\ Implements\ A\ Import\ B\ DefLst;\ rest{:}\mathcal{I}'}$$

B.4 Algorithmic type checking rules

There are two major distinctions between the formal and algorithmic rules for type checking in $\mathcal{LOOM}$. Firstly, note that in almost all cases it is necessary to perform explicit calls to the equivalence or matching algorithms to insure that inferred types for expressions are consistent with their declared types. Secondly, note that there is no explicit notion of subsumption weakening in the algorithm. It is not likely that the formal rules are strictly decidable (or at least deterministic)due to the inclusion of these rules. Instead the notions of subsumption and and weakening are used implicitly within individual rules. To facilitate this, we define a function **mat?eq** as defined below, which handles calls to the equivalence and matching algorithms.

Definition B.4.1 *For types τ and σ, where either type may be of the form $\#\gamma$ for some γ, the function* $\mathbf{mat?eq}(\mathcal{C}, \mathsf{ST}, \tau, \sigma)$ *is defined as follows:*

$$\begin{aligned} \mathbf{mat?eq}(\mathcal{C}, \mathsf{ST}, \#\tau, \#\sigma) \quad &\textit{if} \quad \mathbf{match}(\mathcal{C}, \tau, \sigma) \\ \mathbf{mat?eq}(\mathcal{C}, \mathsf{ST}, \tau, \#\sigma) \quad &\textit{if} \quad \mathbf{match}(\mathcal{C}, \tau, \sigma) \\ \mathbf{mat?eq}(\mathcal{C}, \mathsf{ST}, \tau, \sigma) \quad &\textit{if} \quad \mathbf{equiv}(\mathcal{C}, \tau, \sigma) \end{aligned}$$

and

$\mathbf{mat?eq}(\mathcal{C}, \mathsf{ST}, \{m_2\colon \tau_2; \ldots; m_j\colon \tau_j\}, \{n_1\colon \sigma_1; \ldots; m_1\colon \sigma_i; \ldots; n_j\colon \sigma_j\})$
if $\mathbf{mat?eq}(\mathcal{C}, \mathsf{ST}, \tau_1, \sigma_i)$
and $\mathbf{mat?eq}(\mathcal{C}, \mathsf{ST}, \{m_2\colon \tau_2; \ldots; m_j\colon \tau_j\}, \{n_{i+1}\colon \sigma_{i+1}; \ldots; n_j\colon \sigma_j\})$

The first important difference is the case for handling assignment. Note that in $\mathcal{LOOM}$, a variable may potentially hold objects that match a certain type, as opposed to simply have a certain type. However, if the type of the variable was not explicitly declared as a hash type, we require that the types be identical. We use the function **MatchEq** defined above to enable us to handle both cases with a single rule.

Assn

$$\frac{\mathcal{C}, \mathbf{E} \vdash x\colon \mathit{ref}\ \tau, \quad \mathcal{C}, \mathbf{E} \vdash M\colon \sigma \qquad \mathcal{C}, \mathsf{ST} \vdash \mathbf{mat?eq}(\mathcal{C}, \mathsf{ST}, \sigma, \tau)}{\mathcal{C}, \mathbf{E} \vdash x := M\colon \mathrm{COMMAND}}$$

where both σ and τ may be hash types - that is they each may be of the form $\#\gamma$ for some γ.

The case for function applications is handled differently depending on whether the type of the formal parameter is an exact type or a hash type. Note also even in the exact case, we need to make an explicit call to the equivalence algorithm (via $\mathbf{mat?eq}(\mathcal{C}, \mathsf{ST}, \sigma, \sigma')$) to check that the inferred type of the actual parameter is the same as the declared type of the formal parameter.

$$\textit{FuncAppl} \qquad \frac{\begin{array}{c}\mathcal{C}, \mathbf{E}, \mathsf{ST} \vdash f\colon Func(\sigma)\colon \tau \\ \mathcal{C}, \mathbf{E}, \mathsf{ST} \vdash M\colon \sigma' \\ \mathcal{C}, \mathsf{ST} \vdash \mathbf{mat?eq}(\mathcal{C}, \mathsf{ST}, \sigma, \sigma')\end{array}}{C, E \vdash f(M)\colon \tau}$$

The cases for classes and inherits are similarly different to account for the possibility that the types of instance variables are given as hash types, and hence may receive values of different types. Note that we only present here an algorithmic version of the case for classes, as the inheritance case differs analogously from the formal version.

$$\textit{Class} \qquad \frac{\begin{array}{c}\mathcal{C}^{IV}, \mathbf{E}, \mathsf{ST} \vdash a\colon \sigma' \\ \mathcal{C}^{METH}, \mathbf{E}^{METH}\mathsf{ST} \vdash e_{hid}\colon \tau'_{hid} \\ \mathcal{C}^{METH}, \mathbf{E}^{METH}\mathsf{ST} \vdash e_{vis}\colon \tau'_{vis} \\ \mathcal{C}, \mathsf{ST} \vdash \mathbf{equiv}(\mathcal{C}, \tau_{hid}, \tau'_{hid}) \\ \mathcal{C}, \mathsf{ST} \vdash \mathbf{equiv}(\mathcal{C}, \tau_{vis}, \tau'_{vis}) \\ \mathcal{C}, \mathsf{ST} \vdash \mathbf{mat?eq}(\mathcal{C}, \mathsf{ST}, \sigma, \sigma')\end{array}}{\mathcal{C}, \mathbf{E}, \mathsf{ST} \vdash \texttt{class}(a, (e_{hid}, e_{vis}))\colon \mathbf{ClassType}(\sigma, (\tau_{hid}, \tau_{vis}))}$$

where $\mathcal{C}^{IV} = \mathcal{C} \cup \{\mathbf{MyType} <\# \mathit{ObjectType}\ \tau_{vis}\}$,
$\mathcal{C}^{METH} = \mathcal{C}^{IV} \cup \{\mathbf{SelfType} <\# \mathbf{VisObjType}(\sigma, (\tau_{hid}, \tau_{vis}))\}$,
$\mathbf{E}^{METH} = \mathbf{E} \cup \{\mathit{self}\colon \mathbf{SelfType}, \mathit{close}\colon Func(\mathbf{SelfType})\colon \mathbf{MyType}\}$

Neither **MyType** nor **SelfType** may occur free in $\mathcal{C}$ or $\mathbf{E}$.

σ, τ_{hid}, and τ_{vis} must be record types, while the components of τ_{hid}, and τ_{vis} must be function types.

Appendix C

Complete semantic rules for $\mathcal{LOOM}$

Definition C.0.2 (Environment) *An environment ρ contains bindings of identifiers to values, such as closures, locations, classes etc. The notation $\rho[v/x]$ denotes the binding of value v to identifier f. If identifier f is bound in ρ to value v, we write $\rho(x) = v$.*

See [BSvG95] for a more in depth discussion of the semantic rules of languages such as $\mathcal{LOOM}$.

C.1 Base language semantics

We assume that all terms below have been successfully type checked. In rules where types appear, terms have the same type as was given in the corresponding type-checking rule. A "primed" type (*e.g.*, τ') is generally an abbreviation for the type obtained by applying the substitution from the environment (*e.g.*, $\tau' = \tau_\rho$).

Program:

$$Program \qquad \frac{(Block, \rho_0, s_0) \downarrow (\text{command}, s)}{(\text{program } p;\ Block., \rho_0, s_0) \downarrow (\text{OK}, s)}$$

Declarations:

$$ConstDcls \qquad \frac{(CDclLst, \rho, s) \downarrow (\rho', s')}{(\text{const } CDclLst, \rho, s) \downarrow (\rho', s')}\ ,$$

$$ConstDcl+ \qquad \frac{(CDcl, \rho, s) \downarrow (\rho', s'),\quad (CDclLst, \rho', s') \downarrow (\rho'', s'')}{(CDcl; CDclLst, \rho, s) \downarrow (\rho'', s'')}\ ,$$

$$ConstDcl \qquad \frac{(M, \rho, s) \downarrow (V, s')}{(x = M, \rho, s) \downarrow (\rho[x \mapsto V], s')} ,$$

$$VarDcls \qquad \frac{(VDclLst, \rho, s) \downarrow (\rho', s')}{(\text{var } VDclLst, \rho, s) \downarrow (\rho', s')} ,$$

$$VarDcl+ \qquad \frac{(VDcl, \rho, s) \downarrow (\rho', s'), \quad (VDclLst, \rho', s') \downarrow (\rho'', s'')}{(VDcl; VDclLst, \rho, s) \downarrow (\rho'', s'')} ,$$

$$VarDcl \qquad \frac{(newLoc, s') = (GetNewLoc\ s\ \tau'\ V)}{(x{:}\,\tau, \rho, s) \downarrow (\rho[x \mapsto newLoc], s')} ,$$

where $\tau' = (\tau)_\rho$ and V is the default value for type τ'.

Blocks:

$$Block \qquad \frac{\begin{array}{c}(CDcls, \rho, s) \downarrow (\rho_1, s_1), \quad (VDcls, \rho_1, s_1) \downarrow (\rho_2, s_2), \\ (S, \rho_2, s_2) \downarrow (\text{command}, s_3), \quad (M, \rho_2, s_3) \downarrow (V, s')\end{array}}{(CDcls\ \ VDcls \text{ begin } S \text{ return } M \text{ end}, \rho, s) \downarrow (V, s')} ,$$

Commands:

$$Assign \qquad \frac{(x, \rho, s) \downarrow (Loc, s'), \quad (M, \rho, s') \downarrow (V, s'')}{(x{:=}\,M, \rho, s) \downarrow (\text{command}, s''[Loc \mapsto V])} ,$$

$$Conditional_{true} \qquad \frac{(B, \rho, s) \downarrow (\text{true}, s'), \quad (S, \rho, s') \downarrow (\text{command}, s'')}{(\text{if } B \text{ then } S \text{ else } T \text{ end}, \rho, s) \downarrow (\text{command}, s'')} ,$$

$$Conditional_{false} \qquad \frac{(B, \rho, s) \downarrow (\text{false}, s'), \quad (T, \rho, s') \downarrow (\text{command}, s'')}{(\text{if } B \text{ then } S \text{ else } T \text{ end}, \rho, s) \downarrow (\text{command}, s'')} ,$$

$$while_{true} \qquad \frac{\begin{array}{c}(B, \rho, s) \downarrow (\text{true}, s'), \quad (S, \rho, s') \downarrow (\text{command}, s''), \\ (\text{while } B \text{ do } S \text{ end}, \rho, s'') \downarrow (\text{command}, s''')\end{array}}{(\text{while } B \text{ do } S \text{ end}, \rho, s) \downarrow (\text{command}, s''')} ,$$

while$_{false}$ $$\frac{(B, \rho, s) \downarrow (\text{false}, s')}{(\text{while } B \text{ do } S \text{ end}, \rho, s) \downarrow (\text{command}, s')} ,$$

Sequence $$\frac{(S, \rho, s) \downarrow (\text{command}, s'), \quad (T, \rho, s') \downarrow (\text{command}, s'')}{(S;\ T, \rho, s) \downarrow (\text{command}, s'')}$$

Expressions:

Constant $(c, \rho, s) \downarrow (c, s)$, where c is a constant

Variable $(x, \rho, s) \downarrow (\rho(x), s)$, where x is a variable

Value $$\frac{(M, \rho, s) \downarrow (Loc, s'), \quad s'(Loc) = V}{(\text{val } M, \rho, s) \downarrow (V, s')} ,$$

Function $(\text{function}(v\colon\sigma)B, \rho, s) \downarrow (\langle \text{function}(v\colon\sigma)B, \rho\rangle, s)$,

where σ may be of the form $\#\gamma$.

BdedPolyFunc $(\text{function}(t <\!\# \gamma)B, \rho, s) \downarrow (\langle \text{function}(t <\!\# \gamma)B, \rho\rangle, s)$,

FuncAppl $$\frac{\begin{array}{c}(f, \rho, s^0) \downarrow (\langle \text{function}(v\colon\sigma)B, \rho_f\rangle, s^1), \quad (M, \rho, s^1) \downarrow (V_1, s^2), \\ (B, \rho_f[v \mapsto V_1], s^2) \downarrow (V, s^3)\end{array}}{(f(M), \rho, s^0) \downarrow (V, s^3)} ,$$

where σ may be of the form $\#\gamma$.

BdedPolyFuncAppl $$\frac{\begin{array}{c}(f, \rho, s^0) \downarrow (\langle \text{function}(t <\!\# \gamma)B, \rho_f\rangle, s^1), \\ (B, \rho_f[t \mapsto \sigma_\rho], s^1) \downarrow (V, s^2)\end{array}}{(f[\sigma], \rho, s^0) \downarrow (V, s^2)} ,$$

Record $$\frac{(e_i, \rho, s^i) \downarrow (V_i, s^{i+1}), \quad 1 \le i \le n}{(\{m_1 = e_1\colon \tau_1, \ldots, m_n = e_n\colon \tau_n\}, \rho, s^1) \downarrow (\{m_1 = V_1\colon (\tau_1)_\rho, \ldots, m_n = V_n\colon (\tau_n)_\rho\}, s^{n+1})} ,$$

$$\textit{FieldExt} \qquad \frac{(e,\rho,s)\downarrow(\{m_1 = V_1{:}\,\tau'_1,\ldots,m_n = V_n{:}\,\tau'_n\},s')}{(e.m_i,\rho,s)\downarrow(V_i,s')}\ ,$$

$$\textit{Class} \qquad \frac{(a,\rho',s)\downarrow(V_a,s'),\quad (e,\rho',s')\downarrow(V_e,s'')}{(\mathrm{class}(a,e),\rho,s)\downarrow(\mathrm{class}(V_a,V_e),s'')}\ ,$$

where $\rho' = \rho\backslash\{\mathit{self},\mathit{close},\mathbf{SelfType},\mathbf{MyType}\}$

$$\textit{Object} \qquad \frac{(a,\rho',s)\downarrow(V_a,s'),\quad (e,\rho',s')\downarrow(V_e,s'')}{(\mathrm{obj}(a,e),\rho,s)\downarrow(\mathrm{obj}(V_a,V_e),s'')}\ ,$$

where $\rho' = \rho\backslash\{\mathit{self},\mathit{close},\mathbf{SelfType},\mathbf{MyType}\}$

$$\textit{New} \qquad \frac{\begin{array}{c}(c,\rho,s)\downarrow(\mathrm{class}(\{v_1 = V_1{:}\,\sigma'_1,\ldots,v_n = V_n{:}\,\sigma'_n\},\mathit{Methods}),s^0),\\ (\mathit{newLoc}_i,s^i) = \mathit{GetNewLoc}\ s^{i-1}\ \sigma''_i\ V'_i,\ \text{for } 1\le i\le n,\\ (\mathit{newLoc}_{n+1},s^{n+1}) = \mathit{GetNewLoc}\ s^n\ \mathbf{VisObjType}(\sigma',\tau')\ \mathrm{obj}(\mathit{newIV},\mathit{newMethods})\end{array}}{(\mathrm{new}\ c,\rho,s)\downarrow(\mathit{newLoc}_{n+1},s^{n+1})}\ ,$$

where $\sigma' = \sigma_{\rho'} = \{v_1{:}\,\sigma'_1;\ldots;v_n{:}\,\sigma'_n\}$, $\tau' = \tau_{\rho'}$ for $\rho' = \rho\backslash\{\mathit{self},\mathit{close},\mathbf{SelfType},\mathbf{MyType}\}$,
$\sigma''_i = \sigma'_i[\mathbf{ObjectType}\ \tau'/\mathbf{MyType}]$
$V'_i = V_i[\mathbf{ObjectType}\ \tau'/\mathbf{MyType}])$,
$\mathit{newIV} = \{v_1 = \mathit{newLoc}_1{:}\,\mathit{ref}(\ )\sigma''_1,\ldots,v_n = \mathit{newLoc}_n{:}\,\mathit{ref}(\ )\sigma''_n\}$,
$o' = \mathrm{obj}(\mathit{newIV},\mathit{Methods})$,
if $\mathit{Methods} = \{\langle f_1,\rho_1\rangle,\ldots,\langle f_k,\rho_k\rangle\}$ then $\mathit{newMethods} = \{\langle f_1,\rho'_1\rangle,\ldots,\langle f_k,\rho'_k\rangle\}$
where for $1\le i\le k$,
$\rho'_i = \rho_i[\mathit{self}\mapsto o', \mathit{close}\mapsto \mathit{close}_{(\sigma',\tau')}, \mathbf{SelfType}\mapsto\mathbf{VisObjType}(\sigma',\tau'),$
$\mathbf{MyType}\mapsto\mathbf{ObjectType}\ \tau']$

$$\textit{Close} \qquad \frac{\begin{array}{c}(o,\rho,s)\downarrow(o',s'),\\ (\mathit{newLoc},s'') = \mathit{GetNewLoc}\ s'\ \mathbf{VisObjType}(\sigma',\tau')\ \mathrm{obj}(V_a,\mathit{newMethods})\end{array}}{(\mathit{close}_{(\sigma',\tau')}\ o,\rho,s)\downarrow(\mathit{newLoc},s'')}\ ,$$

where $\sigma' = \sigma_\rho, \tau' = \tau_\rho$ for $\rho' = \rho\backslash\{\mathit{self},\mathit{close},\mathbf{SelfType},\mathbf{MyType}\}$,
$o' = \mathrm{obj}(V_a,\mathit{Methods})$,
if $\mathit{Methods} = \{\langle f_1,\rho_1\rangle,\ldots,\langle f_k,\rho_k\rangle\}$ then $\mathit{newMethods} = \{\langle f_1,\rho'_1\rangle,\ldots,\langle f_k,\rho'_k\rangle\}$
where for $1\le i\le k$,
$\rho'_i = \rho[\mathit{self}\mapsto o', \mathit{close}\mapsto \mathit{close}_{(\sigma',\tau')}, \mathbf{SelfType}\mapsto\mathbf{VisObjType}(\sigma',\tau'),$
$\mathbf{MyType}\mapsto\mathbf{ObjectType}\ \tau']$

$$\textit{Message}\ \frac{(o,\rho,s)\downarrow(Loc,s'),\quad s'(Loc)=\mathrm{obj}(V_a,\{m_1=V_1{:}\,\tau_1',\ldots,m_n=V_n{:}\,\tau_n'\})}{(o\Leftarrow m_i,\rho,s)\downarrow(V_i,s')}\ ,$$

$$\textit{ErrorMessage}\qquad \frac{(e,\rho,s)\downarrow(nil,s')}{(e\Leftarrow m_i,\rho,s)\downarrow(error_{\tau'},s')}\ ,$$

where $\tau'=\tau_\rho$ and τ is the type of $e\Leftarrow m_i$.

$$\textit{InstanceVble}\qquad \frac{(o,\rho,s)\downarrow(\mathrm{obj}(\{v_1=V_1{:}\,\sigma_1',\ldots,v_n=V_n{:}\,\sigma_n'\},V_e),s')}{(o.v_i,\rho,s)\downarrow(V_i,s')}\ ,$$

$$\textit{Inherit1}\ \frac{\begin{array}{c}(c,\rho,s)\downarrow(\mathrm{class}(\{v_1=V_1{:}\,\sigma_1',\ldots,v_{upd}=V_{old}{:}\,\sigma_{upd}',\ldots,v_n=V_n{:}\,\sigma_n'\},V_e),s'),\\ (u,\rho,s')\downarrow(V_{new},s'')\end{array}}{\begin{array}{l}(\text{class inherit } c \text{ modifying } v_{upd};\,(\{v_{upd}=u{:}\,\sigma_{upd}\},\{\}),\rho,s)\downarrow\\ \qquad(\mathrm{class}(\{v_1=V_1{:}\,\sigma_1',\ldots,v_{upd}=V_{new}{:}\,\sigma_{upd}',\ldots,v_n=V_n{:}\,\sigma_n'\},V_e),s'')\end{array}}\ ,$$

where $\rho'=\rho\backslash\{self,close,\mathbf{SelfType},\mathbf{MyType}\}$,
$\sigma_i'=(\sigma_i)_{\rho'}$.

$$\textit{Inherit2}\ \frac{\begin{array}{c}(c,\rho,s)\downarrow(\mathrm{class}(V_a,\{m_1=V_1{:}\,\tau_1',\ldots,m_{upd}=V_{old}{:}\,\tau_{upd}',\ldots,m_n=V_n{:}\,\tau_n'\}),s'),\\ (u,\rho',s')\downarrow(V_{new},s'')\end{array}}{\begin{array}{l}(\text{class inherit } c \text{ modifying } m_{upd};\ (\{\},\{m_{upd}=u{:}\,\tau_{upd}'\}),\rho,s)\downarrow\\ \qquad(\mathrm{class}(V_a,\{m_1=V_1{:}\,\tau_1',\ldots,m_{upd}=V_{upd}{:}\,\tau_{upd}',\ldots,m_n=V_n{:}\,\tau_n'\}),s'')\end{array}}\ ,$$

where $sup=\langle\mathrm{function}(self{:}\,\mathbf{SelfType}).\{m_1=V_1{:}\,\tau_1',\ldots,m_{upd}=V_{old}{:}\,\tau_{old}',\ldots,m_n=V_n{:}\,\tau_n'\},\rho\rangle$
$\rho'=\rho[super\mapsto sup]\backslash\{self,close,\mathbf{SelfType},\mathbf{MyType}\}$
$\tau_i'=(\tau_i)_{\rho'}$.

$$\textit{Inherit3}\ \frac{(c,\rho,s)\downarrow(\mathrm{class}(\{v_1=V_1{:}\,\sigma_1',\ldots,v_n=V_n{:}\,\sigma_n'\},V_e),s'),\quad(u,\rho,s')\downarrow(V_{n+1},s'')}{\begin{array}{l}(\text{class inherit } c\ (\{v_{n+1}=u{:}\,\sigma_{n+1}\},\{\}),\rho,s)\downarrow\\ \qquad(\mathrm{class}(\{v_1=V_1{:}\,\sigma_1',\ldots,v_n=V_n{:}\,\sigma_n',v_{n+1}=V_{n+1}{:}\,\sigma_{n+1}'\},V_e),s'')\end{array}}\ ,$$

where $\rho'=\rho\backslash\{self,close,\mathbf{SelfType},\mathbf{MyType}\}$,
$\sigma_i'=(\sigma_i)_{\rho'}$.

$$\textit{Inherit4}\ \frac{(c,\rho,s)\downarrow(\mathrm{class}(V_a,\{m_1=V_1{:}\,\tau_1',\ldots,m_n=V_n{:}\,\tau_n'\}),s'),\quad(u,\rho',s')\downarrow(V_{n+1},s'')}{\begin{array}{l}(\text{class inherit } c\ (\{\},\{m_{n+1}=u{:}\,\tau_{n+1}\}),\rho,s)\downarrow\\ \qquad(\mathrm{class}(V_a,\{m_1=V_1{:}\,\tau_1',\ldots,m_n=V_n{:}\,\tau_n',m_{n+1}=V_{n+1}{:}\,\tau_{n+1}'\}),s'')\end{array}}\ ,$$

where $\rho'=\rho\backslash\{self,close,\mathbf{SelfType},\mathbf{MyType}\}$,
$\tau_i'=(\tau_i)_{\rho'}$.

C.2 Module semantics

emptyDecLst $$(emptyDecLst) \downarrow^{\mathcal{N}} (\emptyset)$$

DecLst $$\frac{(decLst) \downarrow^{\mathcal{N}} \mathcal{N}}{(f\!:\tau; decLst) \downarrow^{\mathcal{N}} (\mathcal{N} \cup \{f\})}$$

emptyDefLst $$(emptyDefLst, \rho_i, \mathcal{N}, \rho_e, s) \downarrow^{d} (\rho_e, s)$$

DefLst $$\frac{\begin{array}{c} f \in \mathcal{N} \\ (e, \rho_i, s) \downarrow (v, s') \\ (DefLst, \rho_i[v/f], \mathcal{N}, \rho_e[v/A::f], s') \downarrow^{d} (\rho'_e, s'') \end{array}}{((f = e\!:\tau; DefLst), \rho_i, \mathcal{N}, \rho_e, s) \downarrow^{d} (\rho'_e, s'')}$$

DefLst $$\frac{\begin{array}{c} f \notin \mathcal{N} \\ (e, \rho_i, s) \downarrow (v, s') \\ (DefLst, \rho_i[v/f], \mathcal{N}, \rho_e, s') \downarrow^{d} (\rho'_e, s'') \end{array}}{((f = e\!:\tau; DefLst), \rho_i, \mathcal{N}, \rho_e, s) \downarrow^{d} (\rho'_e, s'')}$$

Module Lists

ModLst: $\emptyset$ $$(emptyModLst, \Sigma, \rho_o, s) \downarrow^{\Sigma} (\Sigma, s)$$

$ModLst\colon Int$

$$\frac{\begin{array}{c}\Sigma(B) = (\mathcal{N}_B, \rho_B), \quad (decLst) \downarrow^{\mathcal{N}} \mathcal{N} \\ (ModLst, \Sigma \ \cup \ \{(A, \mathcal{N}, \rho_B \cup \rho_o)\}, \rho_o, s) \downarrow^{\Sigma} (\Sigma', s')\end{array}}{(Interface\ A\ Import\ B\ decLst; ModLst, \Sigma, \rho_o, s) \downarrow^{\Sigma} (\Sigma', s')}$$

$ModLst\colon Imp$

$$\frac{\begin{array}{c}\Sigma(A) = (\mathcal{N}_A, \rho_A), \quad \Sigma(B) = (\mathcal{N}_B, \rho_B) \\ (defLst, \rho_A \cup \rho_B, \mathcal{N}, \emptyset, s) \downarrow^{d} (\rho', s') \\ (ModLst, (\Sigma - \{(A, \mathcal{N}_A, \rho_A)\}) \ \cup \{(A, \mathcal{N}_A, \rho')\}, \rho_o, s') \downarrow^{\Sigma} (\Sigma', s'')\end{array}}{(Module\ Implements\ A\ Import\ B\ defList; ModLst, \Sigma, \rho_o, s) \downarrow^{\Sigma} (\Sigma', s')}$$

$ModLst\colon Program$

$$\frac{\begin{array}{c}\Sigma(B) = (\mathcal{N}_B, \rho_B) \\ (Block, \rho_o \cup \rho_B, s) \downarrow \mathbf{OK}\end{array}}{(Program\ P\ Import\ B\ Abrvs\ Block, \Sigma, \rho_o, s) \downarrow \mathbf{OK}}$$

Appendix D

Example *LOOM* programs

D.1 Sets with efficient intersection

```
Interface IntOrdList;

   OrdListType = ObjectType
      find:func(Integer):bool; --find an element
      first: proc(); --move to the first element (off if empty)
      next: proc(); --move to the next elt (off if at end or off)
      off: func():Boolean; -- is current elt off end of list?
      add: proc(Integer); --add an elt, maintaining ordering
      deleteCur: proc();  -- current is next elt after deleteCur
      contains: func(Integer):Boolean; --is param in list (bsearch)
      getCur: func():Integer --get the current element
   end;

   IntSetType = ObjectType include OrdListType
      remove :proc(Integer);  --remove an element
      intersect: proc(MyType) --receiver contains the intersection
   end;                       --note intersect is destructive!


   OrdListClassType = ClassType ... end;

   OrdListClass: OrdListClassType;

end;
```

```
Interface SetOfInt;

   IntSetType <# ObjectType
      add: proc(Integer);
      remove: proc(Integer);
      contains: func(Integer):Boolean;
      intersect: proc(MyType)
   end;

   function newSet(): IntSetType;

end; -- Interface SetOfInt

Module Implements SetOfInt import IntOrdList;
type

   IntSetType = ObjectType include IntOrdList::OrdListType
      remove :proc(Integer);
      intersect: proc(MyType)
   end;

   ListSetClassType = ClassType include IntOrdList::OrdListClassType;
      methods visible
         remove :proc(Integer);
         intersect: proc(MyType)
      end;

const
   ListSetClass = class inherit IntOrdList::OrdListClass
      methods visible
         procedure remove(elt:Integer) is
            begin
               if find(elt) then deleteCur()
            end;
         procedure intersect(other:MyType) is
            begin
               first();
               other.first();
               while (not off()) and (not other.off()) do
```

```
                if getCur() < other.getCur() then
                   deleteCur()
                elsif getCur() > other.getCur() then
                   other.next()
                else
                   next();
                   other.next()
                end
             end -- while
             while not off() do
                deleteCur()
             end -- while
          end -- function
       end:ListSetClassType;  -- class

   function newSet():IntSetType;
      begin
         return new(ListSetClass)
      end;

end -- Module
```

D.2 Type functions vs. hash types - A comparison

D.2.1 Points and ColorPoints using TFuncs

```
program points_tfunc;
-- note that things of type (#)PointType[measurable] (p,p3 below)
-- are very flexible in what they can take as parameters to eq
-- but are restricted in assignment,
-- whereas things of type #PointType[colorMeasurable] (p4 below)
-- are flexible in what can be assigned to them but restricted in eq

type
measurable = ObjectType
                  getx, gety:func():integer;
             end;
colorMeasurable = ObjectType include measurable
                       getColor:func():integer;
             end;
```

```
PointClassType=TFunc[T<#measurable] ClassType
                        var
                           x, y:integer;
                        methods
                           VISIBLE
                           getx, gety:func():integer;
                           move:proc(integer,integer);
                           eq:func(#T):bool;
                           origin:proc();
                        end;

PointType=TFunc[T<#measurable] ObjectType
                methods
                   getx, gety:func():integer;
                   move:proc(integer,integer);
                   eq:func(#T):bool;
                   origin:proc();
                end;

ColorPointClassType= TFunc[T<#colorMeasurable]
                ClassType include PointClassType[T]
                        var
                           color:integer;
                        methods
                           VISIBLE
                           setColor:proc(integer);
                           getColor:func():integer;
                        end;

ColorPointType=TFunc[T<#colorMeasurable] ObjectType include PointType[T]
                        getColor:func():integer;
                        setColor:proc(integer);
                        end;


const startX = 60 :integer;
      startY = 60 :integer;
      jumpX = 4 :integer;
      jumpY = 4 :integer;
```

```
 PointClass = function(T<#measurable):PointClassType[T]
   begin
   return
   class
   var
           x=0:integer;
           y=0:integer;
   methods
   visible
           getx= function():integer begin return x; end;
           gety= function():integer begin return y; end;
           move= procedure(newX, newY:integer)
                 begin
                   x := newX;
                   y := newY;
                 end;
           eq= function(comparePt:#T):bool
                 var isEqual:bool;
                 begin
                   if ((x = comparePt.getx()) &
                       (y = comparePt.gety()))
                   then isEqual:=TRUE;
                   else isEqual:=FALSE;
                   end;
                   return isEqual;
                 end;
           origin = procedure()
                    begin
                     move(0,0);
                    end;
   end;
   end;

ColorPointClass =
   function(T<#colorMeasurable):ColorPointClassType[T]
   begin
   return
   class inherit PointClass(T) modifying eq;
    var
        color=0:integer;
```

```
          methods
                 visible
              getColor = function():integer begin return color; end;
              setColor = procedure(newC:integer)
                         begin
                           color := newC;
                         end;
              eq= function(comparePt:#T):bool
                         var isEqual:bool;
                         begin
                           isEqual:=FALSE;
                           if (super.eq(comparePt) &
                               (color = comparePt.getColor()))
                           then isEqual:=TRUE;
                           end;
                         return isEqual;
                         end;
           end;
           end;

var
        cp:ColorPointType[colorMeasurable];
        cp2:#ColorPointType[colorMeasurable];
        p:PointType[measurable];
        p2:PointType[colorMeasurable];
        p3:#PointType[measurable];
        p4:#PointType[colorMeasurable];
        x:BOOL;
begin

cp:=new (ColorPointClass(colorMeasurable));
cp2:=new (ColorPointClass(colorMeasurable));

p:=new (PointClass(measurable)); --Legal
--p:=new (PointClass(colorMeasurable)); --Illegal
--p:=new (ColorPointClass(colorMeasurable)); --Illegal

--p2:=new (PointClass(measurable)); --Illegal
p2:=new (PointClass(colorMeasurable)); --Legal
--p2:=new (ColorPointClass(colorMeasurable)); --Illegal
```

```
p3:=new (PointClass(measurable)); --Legal
--p3:=new (PointClass(colorMeasurable)); --Illegal
--p3:=new (ColorPointClass(colorMeasurable)); --Illegal

--p4:=new (PointClass(measurable)); --Illegal
p4:=new (PointClass(colorMeasurable)); --Legal
p4:=new (ColorPointClass(colorMeasurable)); --Legal

x:=cp.eq(cp);
x:=cp.eq(cp2);
--x:=cp.eq(p);          --Illegal
--x:=cp.eq(p2);         --Illegal
--x:=cp.eq(p3);         --Illegal
--x:=cp.eq(p4);         --Illegal

x:=cp2.eq(cp);
x:=cp2.eq(cp2);
--x:=cp2.eq(p);         --Illegal
--x:=cp2.eq(p2);        --Illegal
--x:=cp2.eq(p3);        --Illegal
--x:=cp2.eq(p4);        --Illegal

x:=p.eq(cp);
x:=p.eq(cp2);
x:=p.eq(p);
x:=p.eq(p2);
x:=p.eq(p3);
x:=p.eq(p4);

x:=p2.eq(cp);
x:=p2.eq(cp2);
--x:=p2.eq(p);           --Illegal
--x:=p2.eq(p2);          --Illegal
--x:=p2.eq(p3);          --Illegal
--x:=p2.eq(p4);          --Illegal

x:=p3.eq(cp);
x:=p3.eq(cp2);
x:=p3.eq(p);
```

```
x:=p3.eq(p2);
x:=p3.eq(p3);
x:=p3.eq(p4);

x:=p4.eq(cp);
x:=p4.eq(cp2);
--x:=p4.eq(p);           --Illegal
--x:=p4.eq(p2);          --Illegal
--x:=p4.eq(p3);          --Illegal
--x:=p4.eq(p4);          --Illegal


end.
```

D.2.2 Points and ColorPoints using hash types

```
program points;
--nothing with a pound type below can be sent an eq message, because
--of the rule for message send.
--Things of type PointType can take anything else as a parameter to eq.
--Things of type #PointType can be assigned anything.

type
PointClassType=ClassType
                        var
                           x, y:integer;
                        methods
                           VISIBLE
                           getx, gety:func():integer;
                           move:proc(integer,integer);
                           eq:func(#mytype):bool;
                           origin:proc();
                        end;

PointType=ObjectType
                  methods
                     getx, gety:func():integer;
                     move:proc(integer,integer);
                     eq:func(#mytype):bool;
                     origin:proc();
```

```
                end;

ColorPointClassType= ClassType include PointClassType
                        var
                           color:integer;
                        methods
                           VISIBLE
                           setColor:proc(integer);
                           getColor:func():integer;
                        end;

ColorPointType=ObjectType include PointType
                        getColor:func():integer;
                        setColor:proc(integer);
                        end;


const startX = 60 :integer;
      startY = 60 :integer;
      jumpX = 4 :integer;
      jumpY = 4 :integer;
      PointClass = class
        var
                x=0:integer;
                y=0:integer;
        methods
        visible
                getx= function():integer begin return x; end;
                gety= function():integer begin return y; end;
                move= procedure(newX, newY:integer)
                      begin
                        x := newX;
                        y := newY;
                      end;
                eq= function(comparePt:#mytype):bool
                      var isEqual:bool;
                      begin
                        if ((x = comparePt.getx()) &
                            (y = comparePt.gety()))
                        then isEqual:=TRUE;
```

```
                        else isEqual:=FALSE;
                        end;
                        return isEqual;
                      end;
               origin = procedure()
                        begin
                         move(0,0);
                        end;
        end:PointClassType;

     ColorPointClass = class inherit PointClass modifying eq;
         var
             color=0:integer;
         methods
                visible
             getColor = function():integer begin return color; end;
             setColor = procedure(newC:integer)
                        begin
                          color := newC;
                        end;
             eq= function(comparePt:#mytype):bool
                        var isEqual:bool;
                        begin
                          isEqual:=FALSE;
                          if (super.eq(comparePt) &
                              (color = comparePt.getColor()))
                          then isEqual:=TRUE;
                          end;
                        return isEqual;
                        end;
          end:ColorPointClassType;

var
        cp:ColorPointType;
        cp2:#ColorPointType;
        p:PointType;
        p2:#PointType;
        x:BOOL;
begin
```

```
cp:=new (ColorPointClass);
cp2:=new (ColorPointClass);
p:=new (PointClass);
p2:=new (ColorPointClass); --legal
p2:=new (PointClass);

x:=cp.eq(cp);
x:=cp.eq(cp2);
--x:=cp.eq(p);  --Illegal

 --no legal eq for cp2, since cp2.eq:#MT->bool
--x:=cp2.eq(cp); --Illegal
--x:=cp2.eq(cp2); --Illegal
--x:=cp2.eq(p);         --Illegal
--x:=cp2.eq(p2);        --Illegal

x:=p.eq(p);
x:=p.eq(p2);
x:=p.eq(cp);
x:=p.eq(cp2);

 --no legal eq for p2, since p2.eq:#MT->bool
--x:=p2.eq(p); --Illegal
--x:=p2.eq(p2); --Illegal
--x:=p2.eq(cp); --Illegal
--x:=p2.eq(cp2); --Illegal

end.
```