

UNIVERSITY OF CALIFORNIA
Los Angeles

***Typmix: A Framework For Implementing Modular, Extensible
Type Systems***

A thesis submitted in partial satisfaction
of the requirements for the degree
Master of Science in Computer Science

by

Thomas Anthony Bergan

2007

© Copyright by
Thomas Anthony Bergan
2007

The thesis of Thomas Anthony Bergan is approved.

Jens Palsberg

Todd Millstein

Eddie Kohler, Committee Chair

University of California, Los Angeles

2007

TABLE OF CONTENTS

1	Introduction	1
1.1	Related Work	2
1.1.1	Extensible Compilers	2
1.1.2	Domain-Specific Rule Languages	3
1.1.3	Logic Programming	3
1.1.4	Attribute Grammars	4
1.2	Our approach	4
2	Design and Implementation	7
2.1	Xoc Background	7
2.2	Design Goals	9
2.2.1	Typerules	10
2.2.2	Scoperules	11
2.3	Typerules	12
2.3.1	Evaluation	15
2.3.2	Unification	17
2.3.3	Error Messages	18
2.3.4	Backtracking	19
2.3.5	Binary Relations	20
2.4	Scoperules	20
2.4.1	Default Threading Rules	23
2.4.2	Translation	24
2.4.3	Scope API	24
2.5	Putting It All Together	27
3	Case Study: ML	30
3.1	Damas-Milner	30
3.2	References and Tuples	32
3.3	Data Types and Pattern Matching	34
3.4	Type Qualifiers	35
3.5	Gradual Typing	38

4	Case Study: Java	40
4.1	Featherweight Java	40
4.2	Primitive Types and Control Structures	45
4.3	Generics	46
4.4	Checked Exceptions	48
4.5	Race Detection	50
5	Conclusion	52
	References	53

LIST OF FIGURES

1.1	The complete simply typed lambda calculus with nats and bools.	6
2.1	A fragment of the simply-typed lambda calculus extended with booleans	8
2.2	Typing rules for try and throw in Typmix style	11
2.3	Scoping rules for try and throw	11
2.4	Syntax of Typmix typerules	14
2.5	Translation of logic expressions into xoc statements	16
2.6	Pseudocode for the simply-typed lambda calculus' type judgement	16
2.7	Translation of a single computation rule	17
2.8	The unification API	17
2.9	The backtracking API	19
2.10	Syntax of Typmix scoperules	22
2.11	Default threading rules in pseudocode	23
2.12	Translation of the simply-typed lambda calculus scoperules, part 1	23
2.13	Translation of the simply-typed lambda calculus scoperules, part 2	24
2.14	The Scope API	25
2.15	The Frames API	26
2.16	The simply-typed lambda calculus base language.	28
2.17	“nat” extension module	29
2.18	“bool” extension module	29
2.19	“nat+bool” composition module	29
3.1	Grammar, scoperules, and typerules for Lambda.	31
3.2	References extension	33
3.3	Tuples extension	33
3.4	Grammar extensions for Lambda type constructors and pattern matching	34
3.5	Kinding rules	35
3.6	Typerules and scoperules for pattern matching	36
3.7	Extending pattern matching with tuple support	36
3.8	Grammar extension for type qualifiers	36
3.9	Integer qualifiers	38
3.10	const reference qualifier	39

3.11	Gradual typing extension	39
4.1	Java grammar	41
4.2	Java typing judgements	42
4.3	A sample of Java typing rules	42
4.4	Java scopes	43
4.5	Some of Java's scoping rules	44
4.6	Grammar of primitive types and control structures	45
4.7	Sample typerules and scoperules for primitive types and control structures	46
4.8	Grammar extensions for Java generics	47
4.9	Scoping rules for polymorphic class parameters	47
4.10	Checked exceptions	49
4.11	Type-based race detection	51

LIST OF TABLES

2.1	The Scope API, explained	26
-----	------------------------------------	----

ABSTRACT OF THE THESIS

Typmix: A Framework For Implementing Modular, Extensible Type Systems

by

Thomas Anthony Bergan

Master of Science in Computer Science
University of California, Los Angeles, 2007
Professor Eddie Kohler, Chair

A large body of research has been devoted to designing and developing type systems of varying complexity, power, and purpose. Unfortunately, programmers who wish to use these type systems are at the whim of language designers; relatively few of these ideas have found their way into mainstream programming languages. *Extensible compilers* promise to provide developers the power to customize their languages, including their type systems, as they see fit. While these compilers are indeed powerful, they tend to be large, complex systems that take a significant effort to learn. In this thesis we present Typmix, a modular and easy-to-use framework for implementing type systems that is suitable for inclusion inside an extensible compiler. We describe the two key components of Typmix: an attribute language for declaring *scoping rules*, and a small logic language for declaring *typing rules*. We discuss how these components can be implemented inside an existing extensible compiler, and we demonstrate how they can be used to create and extend type systems by studying their application to both an ML-like language and a Java-like language.

CHAPTER 1

Introduction

This thesis presents a modular and easy-to-use framework for implementing type systems that is suitable for inclusion inside an extensible compiler.

Types are an important programming tool. They serve as self-documentation; unlike comments, compiler checks prevent types from ever being out-of-date. They provide abstraction, through features such as classes and modules. Further, type-based analysis has guided compiler optimizations since its early use in FORTAN [Bac98], which uses types to distinguish numerical representations. Perhaps most importantly, a sound type system guarantees that “well-typed programs do not go wrong” [Mil78]. Thus, types provide a conservative proof of program safety. Largely inspired by the holy grail of provable safety, the research community has invented a multitude of type systems of varying complexity and mechanism, from general frameworks like type classes [WB89] to domain-specific features like type-based race checkers [FF00]. More powerful type systems continue to strengthen the definition of “safety” while increasing programmer expressiveness.

Unfortunately, few of these ideas have made their way into mainstream programming languages. Language designers are forced to please large and diverse user communities, so they rarely make changes that satisfy only a small subset of the larger community. Recent examples from academia which have not made it to the mainstream include type qualifiers [FFA99], protocol checking with labels [ML00] and capabilities [DF01], and bounds checking through dependent types [CHA07, XP98].

Further, developers occasionally find their language’s built-in type system too restrictive. For example, checked exceptions, which are built-in to Java, have been criticized by developers because they are fragile and do not scale well. They were left out of C# for for this reason [Hej03]. A developer may want to use checked exceptions sometimes but not others; currently the only options are all (Java) or none (C#).

For these reasons, Gilad Bracha proposed *pluggable types* [Bra04]. The idea is to make the type system optional so that users can pick and choose which type system they want to use for each project. Like Bracha, we believe that users shouldn’t be bound to the whim of a committee, but instead should have the power to mold their type systems as they see fit. An ambitious developer who desires this power has few options; either he must develop a customized front end [NCH05] or edit an existing compiler [GJS06]. The considerable effort of doing so, however, means that such customizations are rarely attempted.

Developers have not only desired to modify type systems, but also the syntax and semantics of an entire language. Recently, *extensible compilers* have been developed with this exact goal in mind. Ideally, end users can use extensible compilers to mix and match their favorite language extensions,

much in the same way they import libraries today. While certainly powerful and useful, these compilers tend to be large and complex beasts, limiting their use to compiler experts and language designers.

We wish to exploit the power of current extensible compiler technology, but would like a simpler interface for constructing type systems. Typmix is a system that achieves this goal.

1.1 Related Work

In this section, we survey the large body of work relating to type system implementation, and specifically review frameworks for implementing type systems modularly. We first survey extensible compilers, which solve a super-set of the problems that Typmix targets. Following is a survey of frameworks specifically designed for type system extensions. Finally we survey logic programming languages and attribute grammars, both of which provided inspiration for key components of Typmix.

1.1.1 Extensible Compilers

As mentioned earlier, *extensible compilers* grew out of the desire to achieve full language extensibility. An early attempt at such a compiler is CIL [NMR02]. CIL is a modular C front end that was created specifically to make CCured [NCH05], a type system extension, easier to implement. Since then, CIL has been used to implement a variety of type system extensions for C, including Deputy [CHA07] and Clarity [CMM05]. CIL’s modular, visitor-based architecture allows the composition of independent analysis modules, but does not support extensible syntax, meaning that extension authors must still “edit the compiler” to build a type system extension that changes C’s syntax.

Polyglot [NCM03] is an extensible compiler framework for Java which does support extensible syntax. It makes extensive use of design patterns such as visitors and delegates to encourage extensibility. Polyglot is aimed at building monolithic extensions, where an extension produces a new language derived from an existing language. As shown in [CBCon], these extensions cannot be easily composed, and the interface for writing them is heavy-weight. A version of Polyglot ported to the J& language [NQM06] addresses the problem of extension composability. J& uses intersection types to merge multiple classes, such as different extensions to class `Type`. Merge conflicts, such as different versions of the pass schedule, must be resolved manually by a programmer. This precludes arbitrary, dynamic loading of extensions. However, this restriction forces the extension developer to think carefully about how the extensions may conflict; while not a guarantee of correctness, it is still a potentially valuable safeguard.

Xoc [CBCon] is a recent extensible compiler framework that allows extensions to be loaded and composed dynamically at compile time, much like libraries. It also has a simpler interface than Polyglot, meaning that xoc extensions are less heavy-weight than those written for Polyglot. Typmix is built on top of xoc, so we cover it extensively in section 2.1.

1.1.2 Domain-Specific Rule Languages

Recently, partly inspired by Bracha’s arguments for pluggable types, a few domain-specific, rule-based languages for specifying type system extensions have emerged.

Clarity [CMM05] is a framework for defining custom type qualifiers for C. It uses a simple rule language to describe how to inject qualifiers into a base type, and further to define the semantics of a qualifier via invariants. For example, a rule for the `pos` integer qualifier says that addition of two `pos` expressions produces a `pos` expression, and the semantic invariant says that all `pos` expressions are greater than zero. Impressively, Clarity is capable of automatically proving user-defined rules sound with respect to the supplied invariants. However, Clarity’s desire to automatically generate soundness proofs limits its expressiveness.

JavaCOP [ANM06] abandons the soundness guarantees of Clarity, but greatly surpasses Clarity in expressiveness. JavaCOP uses a global-rules approach to specify constraints on syntax trees, much like CADET (discussed later). It includes an ASTLOG [Cre97] inspired pattern-language to help make these rules more concise. The JavaCOP framework hooks into `javac`, applying all user-defined constraint rules after standard Java typechecking. JavaCOP modules can make use of Java 1.5 annotations to express richer type systems than those defined by core Java, but are limited to what is encodable in those annotations. In particular, JavaCOP modules cannot extend the syntax of Java.

TinkerType [LP03] is a framework for playing with type systems. A TinkerType developer constructs a set of rules, annotates each rule with a set of features, and defines a dependency relation on the set of features. Each rule is a set of clauses, where each clause is an uninterpreted string of \TeX or ML code. Given a set of desired features, the TinkerType framework composes all rules belonging to that set of features (and features they depend on, transitively) into a single system. The composition is not a simple union; two features often have conflicting versions of the same rule. When conflicts arise, TinkerType attempts to select the “most specific” version of a rule. For example, there might be two versions of the rule for function application: one with and one without subtyping. The subtyping rule adds a new clause (a subtype check) which refines the previous rule, making it more specific. If TinkerType cannot find a most specific rule, one must be constructed manually. (See [LP03] for a more precise explanation.) Typmix maintains a rule database like TinkerType but does not try to select a single most specific rule. Instead, as detailed in section 2.2, Typmix provides fined-grained control so that pieces of a rule can be defined modularly and composed automatically.

1.1.3 Logic Programming

The Curry-Howard correspondence [How80] states that each type system has an equivalent logic. *Logic programming* languages are a natural framework for encoding of type inference rules. Logic frameworks are commonly used to formalize and prove properties about type systems. Twelf [PS99], for example, is such a framework. An effort is underway to completely formalize and verify Standard ML using Twelf [LCH07]. While powerful, Twelf is purely a formal tool; it cannot be used to generate practical implementations.

Like Twelf, Coq [Coq06] is logic framework frequently used for language formalization. Coq can

translate programs written in its internal language into ML programs. This capability was used to build a certified compiler [Ler06]. The promise that a single framework can generate both a practical implementation and an (assisted) proof of correctness is exciting. However, Coq’s heavy use of dependent types and type-level programming makes for a steep learning curve. Coq has not been widely adopted beyond the research community.

More practically, Odersky’s calculus on derivation trees (CADET) [Ode93] is a specification language built on first-order predicate logic. CADET formalizations are composed of predicate functions that range over the nodes of a syntax tree, and global rules that specify where the predicates must hold. The CADET runtime interprets these predicates to check for conformance. Interestingly, CADET does not use explicit contexts. For example, traditional typing rules carry a set of bound-variable assumptions that is adjusted at declaration nodes before being transported down the syntax tree. In contrast, CADET performs bound-variable resolution using a predicate that searches up the syntax tree, looking for the innermost declaration. Odersky claims that this style of global rules, as opposed to local rules, produces “surprisingly short formal definitions”, but freely admits that it does not lead to efficient implementations.

1.1.4 Attribute Grammars

Attribute grammars (AGs) were proposed by Knuth [Knu68] as a declarative way to specify language semantics. An AG is composed of a set of functions, where each function defines the value of an attribute at a certain syntax node as a function of other attributes on that node or child nodes. Each function can be thought of as a local propagation rule. The order of attribute evaluation is specified implicitly by the dependencies among attribute functions; in other words, the functions are evaluated lazily [Joh87]. Much work has been done to construct efficient AG evaluators and to analyze AGs for circular dependencies.

Two properties of AGs are extremely attractive for our purposes. First, they specify what to compute, but not how to compute it. Second, the composition of two AGs is the simple union of their attributes. To see how this is beneficial towards the construction of modular typecheckers, consider adding C-like goto statements to Java. The namespace of goto labels in C spans an entire function. To emulate this behavior in Java with a traditional typechecker, a special pass would have to be manually inserted to compute the label namespace of a function. An AG-based typechecker would need new attributes, but these attributes could be specified modularly and then unioned with the base typechecker; the new pass is constructed lazily by the AG evaluator. Many AG-based systems have been developed; see [Paa95, VKS07] and citations there.

1.2 Our approach

We want to give end-users the power to modify and extend their languages’ type systems as they see fit. In Typmix, our approach is to design a simple and practical interface that leverages existing technologies while making them easier to use. The foundation of Typmix is xoc, an extensible

compiler. Xoc provides five key features which made Typmix easier to implement. *Extensible grammars*, *extensible functions*, and *extensible data structures* are the core technical mechanisms of xoc. They suffice to enable complete extensibility of the base compiler, while doing so in a modular way such that independent extensions can be composed. *Syntax patterns* and *lazy attributes* make these mechanisms easier to use by abstracting away unimportant compiler details, such as the internal representation of syntax trees and the global pass structure. Xoc extensions are written in a standard procedural manner, augmented with these five techniques. Our experience suggests that this style is optimized for syntax rewriting; type systems implemented with xoc tend to require verbose redundancies. Typmix seeks to provide a simpler and more natural interface for type system development in xoc.

Typmix introduces two mini-languages on top of the xoc core: *scoperules* and *typerules*. The next chapter explains why the separation of these components is an important design decision. In fact, a previous version of Typmix did not make this separation and was much harder to use because of it.

The *scoperules* language, inspired heavily by attribute grammars, uses local rules to concisely declare how scopes propagate. We opted for an explicit local-rules style as opposed to the global-rules style of CADET for two reasons. First, there is a simple and efficient encoding of the *scoperules* language into xoc attributes. Second we believe that explicit scopes provide a more obvious and natural abstraction than CADET's global rules.

The *typerules* language is a mini-logic language, inspired heavily by JavaCOP and by the traditional inference rule formalization of type systems. The language consists of judgements, which relate syntax nodes to values (for example, an expression to its type). Judgements are defined by a set of syntax-directed inference rules which are interpreted in a first-order logic. The unification algorithm for this logic is extensible, so that the framework can be tailored to the requirements of the type system being implemented.

Though we believe that Typmix reaches our goals of modularity and ease of use, it has limitations. Most importantly, Typmix makes no correctness guarantees. This is important in two ways. First, although independent Typmix extensions can be composed, we cannot know whether the composition will work correctly. This is true even when it seems that the extensions *should* be semantically compatible; a poor implementation decision which has no adverse effects in isolation could break the composition. Mitigating this somewhat, a developer always has the option to construct an extension which is the manual composition of two extensions. Second, although Typmix's *typerules* and *scoperules* are declarative, and seem amenable to analysis, Typmix provides no built-in theorem prover or interface to an off-the-shelf verifier. Extension authors are responsible for the correctness of their code. We leave automatic and assisted proofs of correctness for future work.

Figure 1.1 presents a complete implementation of the simply typed lambda calculus in Typmix, intended to give a high-level flavor of the system. Chapter 2 covers the design and implementation of Typmix in depth. Chapter 3 presents our first case study: using Typmix to implement and extend the type system of an ML-like language. Chapter 4 presents a second case study for a Java-like language. Finally, Chapter 5 concludes.

```

grammar Simple
{
  %ignore /[\t\n]+/;
  %left App;
  %right Arrow;
  %priority App > Abs;

  expr: "(" expr ")"          [ paren ]
      | expr expr             [ App ]
      | "fun" var ":" type "." expr [ Abs ]
      | "if" expr "then" expr "else" expr [ Abs ]
      | var
      | "true" | "false"
      | "0";

  type: "(" type ")"      [ paren ]
      | type "->" type   [ Arrow ]
      | "Bool"
      | "Nat";

  var: /[a-z][A-Za-z0-9_]*;/
}

scoperules
{
  vars: Scope<Simple.type>;

  globals {
    vars = scopebindlist([(`var{succ}, `type{Nat->Nat}),
                          (`var{pred}, `type{Nat->Nat}),
                          (`var{iszero}, `type{Nat->Bool})],
                          new Scope());
  }

  ~expr{fun \x:\T. \e} {
    e.vars@in = scopebind(x, T, default);
    e.vars@out = term.vars@in;
  }
}

typerules
{
  judgement type: (term: Simple.expr) "::" Simple.type;

  ~expr{\x::var} :: term.vars[x];
  ~expr{true}    :: `{Bool};
  ~expr{false}   :: `{Bool};
  ~expr{0}       :: `{Nat};

  ~expr{fun \x:\T1. \e} :: `{T1 -> T2} { e :: T2 }
  ~expr{\e1 \e2}      :: T { e1 :: {T2 -> T} && e2 :: T2 }

  ~expr{if \e then \t else \f} :: T { e :: {Bool} && t :: T && f :: T }
}

```

Figure 1.1: The complete simply typed lambda calculus with nats and bools.

CHAPTER 2

Design and Implementation

Typmix was designed to be a modular and easy-to-use framework for implementing type systems. This chapter covers the design and implementation of Typmix in detail. First, we overview the xoc compiler framework, which is the foundation of Typmix. Next is a high-level discussion of the Typmix design. Finally, all the details of *typerules* and *scoperules* are revealed.

2.1 Xoc Background

The xoc compiler framework is detailed in [CBCon]. Xoc is written in a statically-typed procedural language, similar to C but extended with features designed to make extensible compilers easier to write. Five features of that language are relevant to the rest of this thesis: grammars, syntax patterns, extensible functions and data structures, and on-demand computation of attributes. Most important to Typmix are syntax patterns and on-demand attributes, but for completeness all features are briefly described here.

Grammars and syntax patterns Grammars in xoc are defined using context free parsing rules, which are compiled into a GLR parser at run time. Grammars are extensible. For example, figure 2.1 shows a fragment of a grammar for the simply-typed lambda calculus, and an extension to that grammar which adds booleans.

Grammars are exposed directly as xoc data types such as `Ast`, `Ast.expr`, and `Ast.type`.¹ The types exist in an implicit subtyping heirachy such that `Ast` is a supertype of `Ast.expr` and `Ast.type`. The type `Ast` refers to any syntax node, making it the natural `Top` of this subtyping hierarchy. Syntax types can be further refined by grammar, like `Simple.expr`, or by explicitly named rules like `Simple.type.Bool` (which derives from the `<Bool>` rule on line 22).

The format of AST nodes is not exposed directly. Rather, AST nodes are manipulated using concrete syntax patterns. A *destructuring* expression like

$$f \sim \text{expr}\{\text{fun } \backslash x:\backslash t. \backslash e\}$$

matches AST node `f` with a syntax pattern. Upon success, the slots `x`, `t`, and `e` are bound to matching subtrees in `f`. A *restructuring* expression like

¹The actual xoc types are prefixed with `ptr`, as in `ptr Ast`, but in this thesis we elide `ptr` for brevity.

```

1  grammar Simple {
2    ...
3    %left App;
4    %right Arrow;
5    %priority App > Abs;
6
7    expr: "(" expr ")"           [ paren ]
8         | expr expr             [ App ]
9         | "fun" var ":" type "." expr [ Abs ]
10        | var;
11   type: "(" type ")"           [ paren ]
12        | type "->" type;       [ Arrow ]
13
14   ...
15 }
16
17 extend grammar Simple
18 {
19   expr: "if" expr "then" expr "else" expr
20        | "true" | "false";
21
22   type: <Bool> "Bool";
23 }

```

Figure 2.1: A fragment of the simply-typed lambda calculus extended with booleans

$$\text{\code{`expr\{fun \x:\t. \e\}}$$

rebuilds an AST node by substituting the slots `x`, `t`, and `e` into the syntax pattern. Slots are typed, and these types are checked so that syntax patterns are always well-formed. For example, the pattern `{fun x:Bool.\e}` is clearly malformed when `e` has type `Simple.type`. All such errors are detected statically. Slot types can be declared explicitly using syntax like `\e : expr`. If not declared, they are inferred; refer to [CBCon] for details. Finally, the `[paren]` annotation (lines 7 and 11) shows `xoc` how to automatically eliminate parentheses so that `{succ (0)}` and `{succ 0}` have the same meaning (i.e., they will both be matched by the pattern `{succ 0}`).

Tree traversals using just syntax patterns can be verbose. Consider, for example, a `subst` function that substitutes one kind of subtree for another (such as a type for a type variable). If syntax patterns are the only means of inspecting the syntax tree, `subst` would need to exhaustively match against all possible forms. Further, `subst` would become out-of-date whenever the grammar is extended. Other compilers package this capability into an extensible, reusable visitor framework. This approach still requires substantial amounts of boilerplate code. `Xoc` eliminates this boilerplate code by providing a library of generic traversals and transformations, a la *Stratego* [Vis04]. Examples of library functions include `subst` and `rewrite_topdown`. Two primitive functions make this library possible: `astsplit` returns the subtrees of an AST node, and `astjoin` rebuilds an AST node using new subtrees. Notably, `Typmix` takes advantage of this library to provide a generic unification framework (section 2.3.2).

Extensible functions All `xoc` functions are extensible. For example, say there is a function `isvalue` which determines whether an expression is a syntactic value. The following demonstrates how to ex-

tend that function to understand the boolean syntax from figure 2.1:

```
1  extend fn
2  isvalue(expr: Ast.expr): bool
3  {
4      if(expr ~ {true} || expr ~ {false})
5          return true;
6      return default(expr);
7  }
```

This definition replaces the old definition of `isvalue`, but can refer to the old definition in the body of the function using the special name `default`. Thus, extensible functions encode a chain of responsibility.

Extensible data structures Xoc has C-like struct types which can be extended with new fields. Struct extensions are locally scoped so that independent xoc extensions do not accidentally refer to other extension's fields. They are declared by prepending the `extend` keyword, like:

```
1  extend struct Foo {
2      ...
3  }
```

On-Demand Attributes A xoc attribute is a property attached to an AST node. Attributes are invoked like fields, but are defined like functions. Like functions, they can be extended. For example, `isvalue` might instead have been defined as an attribute. It could then be invoked using `e.isvalue`, where `e` is an AST of type `Ast.expr`, and could be extended as below:

```
1  extend attribute
2  isvalue(expr: Ast.expr): bool
3  {
4      if(expr ~ {true} || expr ~ {false})
5          return true;
6      return default(expr);
7  }
```

An attribute can invoke other attributes, including those on subtrees and those on the special neighbors `parent`, `prev`, and `next` (defined in section 2.4.1). Crucially, attributes are computed on-demand and then cached. We show later why on-demand, or *lazy*, evaluation of attributes is important.

2.2 Design Goals

This section motivates important goals and design decisions, and specifically explains why it was necessary to separate Typmix into two languages, *typerules* and *scoperules*.

2.2.1 Typerules

Type system designers formalize type systems using logical inference rules, which are both standard and concise. They therefore might seem like an ideal tool for describing modular and extensible type systems. However, a framework built entirely on inference rules has problems.

Consider, for example, a Java-like language with exceptions. An inference rule encoding of this language might use an OK judgement for checking statements. Here are possible rules for `try` and `throw`:

$$\frac{\Gamma \vdash s_1 :: \text{OK} \quad \Gamma, x : T \vdash s_2 :: \text{OK}}{\Gamma \vdash \text{try } s_1 \text{ catch}(T \ x) \ s_2 :: \text{OK}} \quad \text{OK-TRY1}$$

$$\frac{\Gamma \vdash e : T \quad T <: \text{Exception}}{\Gamma \vdash \text{throw } e :: \text{OK}} \quad \text{OK-THROW1}$$

Above the bar are preconditions, or *premises*, and below the bar is the *conclusion*. Each clause is prefixed by a set of assumptions; in this case, Γ gives the bindings from variables to types. Note how the scoping of program variables is encoded in these rules by modifying Γ appropriately in each clause. Now consider adding checked exceptions to this language (changes highlighted):

$$\frac{\Delta, T; \Gamma \vdash s_1 :: \text{OK} \quad \Delta; \Gamma, x : T \vdash s_2 :: \text{OK}}{\Delta; \Gamma \vdash \text{try } s_1 \text{ catch}(T \ x) \ s_2 :: \text{OK}} \quad \text{OK-TRY2}$$

$$\frac{\Delta; \Gamma \vdash e : T \quad T <: \text{Exception} \quad \exists C \in \Delta. T <: C}{\Delta; \Gamma \vdash \text{throw } e :: \text{OK}} \quad \text{OK-THROW2}$$

A few things stand out. First, notice that the assumption Δ , which represents the set of exceptions which will be caught by some enclosing statement, has been added to the OK judgement. Such a simple change required changing *every* clause in *every* rule! This is easy to fix: we make the assumption environment an extensible data structure, call it *Env*, and refer to *Env*. Γ and *Env*. Δ instead of Γ and Δ .

More problematic is OK-TRY2. Only a small change was made— Δ is updated with T in the assumptions for s_1 —but the rest of rule OK-TRY1 had to be copied anyway! Clearly, this is not modular. The problem is that these inference rules are trying to do too much. This is solved in Typmix by separating the “scoping” from the “typing”. The task of propagating environment changes is moved to a dedicated *scoperules* language, and the task of declaring typing constraints is left to the (now simpler) *typerules* language.

A final problem is witnessed above by OK-THROW2, where a new clause is added to the otherwise unchanged OK-THROW1. The key to maintaining modularity in cases like this is to allow incremental update of existing rules. In this example, the new clause would be declared independently, but added in conjunction with the existing clauses. The details of this mechanism are left for section 2.3.

$\frac{Env \vdash s_1 :: \text{OK} \quad Env \vdash s_2 :: \text{OK}}{Env \vdash \text{try } s_1 \text{ catch}(T x) s_2 :: \text{OK}}$	OK-TRY3
$\frac{Env \vdash e : T \quad T <: \text{Exception}}{Env \vdash \text{throw } e :: \text{OK}} \quad \frac{\exists C \in Env.\Delta. T <: C}{Env \vdash \text{throw } e :: \text{OK}}$	OK-THROW3

Figure 2.2: Typing rules for try and throw in Typmix style

$$\begin{aligned} \text{try } s_1 \text{ catch}(T x) s_2 : \\ s_2.Env.\Gamma &= Env.\Gamma, x : T \\ s_1.Env.\Delta &= Env.\Delta, T \end{aligned}$$

Figure 2.3: Scoping rules for try and throw

The key insight of these techniques is that the granularity of primitives provided by Typmix directly affects modularity. It is important to minimize granularity of language primitives where possible in order to maximize modularity. Figure 2.2 shows the typing inference rules for `try` and `throw` in Typmix style.

2.2.2 Scoperules

Typmix’s *scoperules* language is responsible for describing changes to Env . It does so using a set of local rules. These rules are fine-grained, so that scoping rules do not suffer from the same modularity problems discussed earlier. Figure 2.3 shows scoping rules to match the typing rules from figure 2.2.

Typmix’s scoperules language is in fact an *attribute grammar* language. Each element of the assumption environment Env is an attribute, and rules for attribute propagation are syntax-directed. The first rule in figure 2.3 shows how to update Γ for the catch block, and the second rule shows how to update Δ for the try block. These rules are exactly the same as those in OK-TRY2, only now they are separated from the typing rules so that they can be changed independently.

An important property of attribute grammars is that a global pass scheduler is not required; the pass schedule is lazily constructed at run time by evaluating the rules of the attribute grammar on demand. In this example, the attribute Δ is not computed until it is explicitly requested by OK-THROW3. This further improves modularity and extensibility. To see why, consider the alternative of a global pass scheduler: each extension would need to plug its changes into the scheduler, and extensions written separately would need to have their changes composed manually. With lazy attributes, this happens automatically at run time.

Typmix scoperules are evaluated using `xoc` attributes, which provide the needed laziness. This mechanism and other details of the scoperules language are discussed in section 2.4.

2.3 Typerules

The typerules language is a language for writing type inference rules. Much like other first-order logic programming languages, the typerules language evaluates queries by searching a database of inference rules. Typerule queries are judgements of the form $\vdash \textit{term} \textit{op} \textit{result}$, where *term* is a syntax tree, *op* is a relational operator, and *result* is an arbitrary xoc data type (typically another syntax tree or a boolean). Both *term* and *op* are inputs to the query, and *result* is its output. Backwards execution modes are not supported. Like Prolog, queries are executed by searching a database of inference rules in depth-first search order. Unlike Prolog, the search algorithm is terminating because the typerules language does not support arbitrary non-determinism, as explained later.

As discussed in the previous section, it must be possible to extend typing rules modularly. The typerules language provides two mechanisms for doing so. First, *annotation rules* can be used to compute a more precise type than that computed by a base typing rule. One example of a more “precise” type is a qualified type, as in type-qualifier frameworks. Second, *restrict rules* can be used to specify additional constraints on a computed type. For example, the new throws clause from OK-THROW3 can be added using a restrict rule.

Finally, two aspects of the implementation separate typerules from other logic programming languages such as Prolog. First, the unification algorithm is extensible. This allows type system implementors to tailor the unification algorithm to their language. Second, typerules can invoke arbitrary xoc functions which are executed blindly (they are not interpreted by the logic). To support this, the depth-first search’s backtracking step must be extensible, so that global state manipulated by these function can be manually saved and restored.

This section introduces the typerules language by example, and then details its implementation.

Simply-typed lambda calculus Our example will be the simply-typed lambda calculus with nats and booleans. This language is presented in full in figure 1.1.

```
1 typerules
2 {
3   judgement type: (term: Simple.expr) "::" Simple.type;
4
5   ~expr{true}    :: `Bool;
6   ~expr{false}  :: `Bool;
7   ~expr{0}      :: `Nat;
8   ~expr{\x::var} :: term.vars[x];
9
10  ~expr{fun \x:\T1. \e} :: `{\T1 -> \T2} { e :: T2 }
11  ~expr{\e1 \e2}      :: T          { e1 :: {\T2 -> \T} && e2 :: T2 }
12
13  ~expr{if \e then \t else \f} :: T          { e :: {Bool} && t :: T && f :: T }
14 }
```

Line 3 declares a new judgement named `type`. It relates expressions to types (`Simple.expr` to `Simple.type`) using the operator “`::`”. As explained later, judgements are implemented as a xoc

attribute. Thus, to query this judgement simply invoke its attribute, as in `e.type` where `e` has type `Simple.expr`.

The operator “`::`” is added to Typmix’s grammar of expressions as a new infix operator. It cannot be a symbol predefined by `xoc`, but it may be overloaded by other judgements. For example, we might add a kinding judgement from types to kinds (say, `Simple.type` to `Simple.kind`) which also uses the “`::`” operator. Uses of this operator are disambiguated by the type of their left-hand term. For example, the expression `a :: b` would resolve to the type judgement when `a` is a `Simple.expr` and to the kind judgement when `a` is a `Simple.type`.

Lines 5–7 define the first three inference rules, which are actually axioms that give types for constant expressions. The fourth rule (line 8) uses the `vars` scope on `term`, which is the expression being typed (scopes are explained in section 2.4.3). In this case, `term` matches the pattern `expr { \x : var }`. The name `term` is declared by the judgement declaration at line 3. For consistency, this thesis always uses the name `term` in that place. The final three rules use boolean preconditions, which are written in braces. This syntax is “upside-down” relative to traditional inference rules. For example, the rule for function application (line 11) is exactly equivalent to the more familiar:

$$\frac{\vdash e_1 :: T2 \rightarrow T \quad \vdash e_2 :: T2}{\vdash e_1 e_2 :: T}$$

Each “`::`” expression in the premise is interpreted by the logic. For example, the expression `e1 :: { \T2 -> \T }` on line 11 says that `e1.type` should match the pattern `{ \T2 -> \T }`, where `T2` and `T` are metavariables in the logic. The scope of a metavariable is the entire inference rule, including the premise and the result. Meta-variables range over syntax trees of certain types and are bound by unification. The expression `e1 :: { \T2 -> \T }` is implemented as a call to `unify`; the details are left for section 2.3.2.

Finally, note that the assumptions to this judgement (*Env*, as we called them in the previous section) are carried implicitly as attributes. `term.vars` is one such example. This is explained completely in section 2.4.

Typrules syntax The full syntax of the `typerules` language is given in figure 2.4.

The optional `restrict` qualifier for inference rules is explained shortly. The six expressions shown in figure 2.4 are the only ones interpreted by `typerules` logic. Other `xoc` expressions are executed ignorantly. There are three expressions we have not yet seen: `forall`, `exists`, and `where`. The expression `where(e1)e2` is equivalent to `(e1&&e2) || !e1`. The quantifier `forall(p in S){e}` has its conventional meaning: for all objects in *S* that match pattern *p*, the expression *e* must be true. The set *S* can be a `xoc` list, a `xoc` map (hash table), or a scope (see section 2.4). The quantifier `exists` is analogous.

Annotate rules The examples seen so far show *computation rules*. Such rules compute a term’s type as a function of its subterms and its environment. As mentioned earlier, it is sometimes useful

```

Top ::= typerules { T* }
T   ::= judgement name : (name1 : type1) “op” type2
      | pattern op expr ;
      | pattern op expr { expr }
      | restrict pattern op pattern ;
      | restrict pattern op pattern { expr }
expr ::= forall( pattern in expr ) { expr }
      | exists( pattern in expr ) { expr }
      | where( expr ) { expr }
      | expr op pattern
      | expr && expr
      | expr || expr
      | other usual xoc expressions

```

Figure 2.4: Syntax of Typmix typerules

to *annotate* a computed type, making it more precise. For example, consider using the `pos` numeric qualifier to annotate positive numbers [CMM05]. The addition of two `pos` expressions produces a `pos` expression. A rule for applying `pos` to addition expressions might look like:

```

~expr{\a + \b} :: {pos \T}   { term :: T && a :: {pos \A} && b :: {pos \B} }

```

This rule uses the self-recursive query `term::T`, which states that `term` should be “partially typed” to type `T`. The rule then shows how to annotate the partial type `T` with a more complete type `pos T`. These sorts of rules are *annotation rules*. The set of annotation rules is the set of all rules which make a self-recursive query, as does the `pos` rule above. They are only invoked after a computation rule has returned a “partial” type, and are invoked up to a fixed point. There can in fact be multiple fixed points. Typmix will find one fixed point and stick to it; the other fixed points are never tried (see section 2.3.4 for why this is so). Further, the fixed point may not exist, at least in the conventional meaning of “fixed point”. For example, the following erroneous annotation rule seems to add a `neg` qualifier ad infinitum:

```

~expr{\a + \b} :: {neg \T}   { term :: T }

```

Typmix does not apply any single rule (such as the one above) more than once per “fixed point”. Thus, the above rule will only add one `neg` qualifier, not an infinite number. This is done to guarantee that typerule evaluation is always terminating. However, this means that Typmix’s evaluation strategy is not complete, in the conventional sense. We have not found this to be a problem in practice since annotation rules typically work like the simple `pos` rule above.

Restrict rules Frequently, type system extensions want apply a restriction onto a term that has already been fully typed. (For example, most of the extensions implemented with JavaCOP have this property [ANM06].) To support this common case we use *restrict rules*. Restrict rules are preceded by the `restrict` qualifier, as in:

```
restrict pattern :: rpattern { precondition }
```

A restrict rule is not invoked unless a complete, annotated type has been computed which matches `rpattern`. Once invoked, the `precondition` must be satisfied or else the entire query fails. This is roughly equivalent to the following annotation rule template, except that restrict rules are not invoked until an annotation fixed point has been reached:

```
1 pattern :: `rpattern` {
2   where(term :: rpattern)
3     precondition
4 }
```

As a practical example, here's how the checked exceptions extension from rule OK-THROW3 (section 2.2) might be implemented using `restrict` (where the boolean `true` means OK):

```
1 restrict
2 ~stmt{throw \e;} :: true {
3   where(e :: T)
4     exists(C in term.CaughtExceptions)
5       issubtype(T, C)
6 }
```

2.3.1 Evaluation

Inference rules are encoded directly into `xoc` attributes, so that attribute evaluation is logic evaluation. Translation of the logic language into attributes is summarized by the rewrite rules in figure 2.5.

A logical expression is translated into a sequence of `xoc` statements. In the base case, a pure expression (such as a function call, a constant, or a judgement query like $e :: T$) is checked for false using an `if` statement. The `fail` branch emits an error message (see section 2.3.3) and then backtracks. The other five cases use a checkpointing API to implement a depth-first search, where the `checkpoint` function acts like `set jmp`. The full checkpointing API is described in section 2.3.4, and is summarized in figure 2.9.

Each judgement is translated into a `xoc` attribute, as shown in figure 2.6. Evaluation proceeds directly: a partial type is computed, the type is annotated, and then restriction rules are validated. The first two steps, `compute` and `annotate`, perform a search through the inference rule database to generate a full type. The final step, `restrict`, checks all restriction rules in any order. Each rule is translated to a small function and linked into the database. The template for computation rules is shown in figure 2.7; annotation and restriction rules are similar.

$[[e]]$	\equiv	<code>if(!e) fail;</code>
$[[e_1 \ \&\& \ e_2]]$	\equiv	<code>[[e1]] ; [[e2]]</code>
$[[e_1 \ \ e_2]]$	\equiv	<code>CP := pushcheckpoint();</code> <code>if(CP.restarted) {</code> <code> popcheckpoint(CP);</code> <code> [[e2]]</code> <code>} else {</code> <code> [[e1]];</code> <code> popcheckpoint(CP);</code> <code>}</code>
$[[\text{where}(e_1)\{ e_2 \}]]$	\equiv	<code>CP := pushcheckpoint();</code> <code>if(CP.restarted)</code> <code> popcheckpoint(CP);</code> <code>else {</code> <code> [[e1]];</code> <code> popcheckpoint(CP);</code> <code> [[e2]]</code> <code>}</code>
$[[\text{forall}(p \text{ in } S)\{ e \}]]$	\equiv	<code>for(o in S)</code> <code> if(o matches p) [[e]]</code>
$[[\text{exists}(p \text{ in } S)\{ e \}]]$	\equiv	<code>ok := false;</code> <code>for(o in S)</code> <code> if(o matches p) {</code> <code> CP := pushcheckpoint();</code> <code> if(CP.restarted) {</code> <code> popcheckpoint(CP);</code> <code> continue;</code> <code> } else {</code> <code> [[e]];</code> <code> popcheckpoint(CP);</code> <code> ok = true; break;</code> <code> }</code> <code> }</code> <code>if(!ok)</code> <code> fail;</code>

Figure 2.5: Translation of logic expressions into xoc statements

```

1 attribute
2 type(term: Simple.expr): Simple.type
3 {
4   result = compute(term);
5   if(!result)
6     fail;
7   result = annotate(term, result);
8   if(!restrict(term, result))
9     fail;
10  return result;
11 }

```

Figure 2.6: Pseudocode for the simply-typed lambda calculus' type judgement

```

1 fn(term: Simple.expr): Simple.type
3 {
4   if(term ~ pattern) {
5     [[ precondition ]]
5     return result;
6   }
7   fail;
8 }

```

Figure 2.7: Translation of a single computation rule

```

unify      : (Ast, Ast, UnifyState) -> bool
unifystruct : (Ast, Ast, UnifyState) -> bool
unifyvar   : (LogicalVar, Ast, UnifyState) -> bool
unifyerror : (UnifyState) -> bool

extensible struct LogicalVar {
  unified : Ast;
}

```

Figure 2.8: The unification API

There may be multiple ways to compute an initial result (computation rules) or a fixed point (annotation rules). Currently, Typmix’s backtracking mechanism is not powerful enough to try all possibilities. Instead, the first computation rule that succeeds and the first fixed point that is found become the final result. Reasons for this limitation are discussed in section 2.3.4.

2.3.2 Unification

As mentioned, each judgement query such as $e :: \{\backslash T2 \rightarrow \backslash T\}$ is implemented as a call to `unify`, which is Typmix’s extensible unification algorithm. The interface to this unification algorithm is summarized in figure 2.8.

The `unify` function attempts to unify two syntax trees. In our example, it will attempt to unify `e.type` with the pattern $\{\backslash T2 \rightarrow \backslash T\}$. Importantly, `unify` needs to know which kinds of AST nodes are logical metavariables, and which are actual syntax. For this it uses the attribute `logicalvar`, which returns a data structure called `LogicalVar` for any AST node which should be treated as a logical metavariable. In our example, the pattern includes two special AST nodes $\backslash T2$ and $\backslash T$, called *slots*. By default, `logicalvar` treats all slots as metavariables. This makes patterns such as the above work automatically. It is sometimes useful to implement `logicalvar` for other kinds of AST nodes. For example, an ML-like language with explicitly-named type variables would implement `logicalvar` for said type variables, so that Typmix’s unification algorithm would treat them as logical metavariables as well.

The `unify` function actually dispatches to two other functions, `unifystruct` and `unifyvar`, which do all the real work. These two functions are intended to be extended by a “real” type system, but provide a default implementation which covers the common case.

`unifyvar` is used if either syntax tree is a logical metavariable. The default implementation simply binds the metavariable to the other syntax tree by assigning to `LogicalVar.unified`. It does not do an occurs check. `unifystruct` is used when neither syntax tree is a metavariable. The default implementation of `unifystruct` checks for structural equality. This is done by dynamically comparing the types of both syntax trees (for example, `Ast.expr` vs `Ast.type`) and then recursively unifying the subtrees, where the subtrees are extracted generically using `astsplit`.

So, the default unification algorithm provided by `Typmix` is structural equality and variable binding without an occurs check. This is all that is needed to implement simple languages. For example, our presentation of the simply-typed lambda calculus in figure 1.1 uses the default versions of `unifystruct` and `unifyvar`, without modification. Of course, richer languages (such as those described in Chapters 3 and 4) require a more specialized unification algorithm. More complex unification algorithms could be supported as `Typmix` libraries, but we decided against this. Instead, language designers are expected to extend `unify` to implement a more complex algorithm where necessary.

The `UnifyState` data structure and the `unifyerror` function shown in figure 2.8 are used to support error messages, which are discussed next.

2.3.3 Error Messages

Good error messages are obviously an important requirement of real-world type checkers. It is possible to manually emit error and warning messages in `Typmix` typerules using the library functions `error` and `warn`. Each has two parameters, the program location (where the error occurred) and the message to emit. Each is a boolean function: `warn` returns true and `error` returns false. The typical usage of `warn` is `where(e) warn()`, which emits a warning (but does not fail) when `e` holds. The typical usage of `error` is `e || error()`, which emits an error and fails whenever `e` does not hold.

Applied liberally, this technique quickly becomes verbose. To combat this verbosity, `Typmix` automatically emits an error message in three common cases. First, as mentioned above, the function `unifyerror` is called whenever unification fails. It emits a “unification error” at the program location specified by `UnifyState`, which is automatically managed by `Typmix`.

Second, an error message is automatically emitted anytime a scope lookup fails. Currently this message is not customizable. For example, `Typmix` always says “unknown variable X” even when it would make more sense to say “unknown class X”.

Thirdly, an error message is automatically emitted when a call to a `xoc` function fails. For example, failure of `issubtype(A, B)` would emit an error message like “issubtype failed for A and B”. Like scope error messages, these messages are not customizable.

For simple languages, such as the simply-typed lambda calculus from figure 1.1, these defaults work well. Unfortunately, they do not work very well for more complicated languages which require more detailed error messages. In future work, we may explore how to generate better error messages without requiring verbose, manual calls to `error`.

```

pushcheckpoint : () -> Checkpoint
popcheckpoint  : (Checkpoint) -> ()
backtrack     : () -> ()

extensible struct Checkpoint {
    restarted : int;
}

```

Figure 2.9: The backtracking API

2.3.4 Backtracking

As previously mentioned, Typmix executes a typing query using a depth-first search through a database of inference rules. When a clause fails (evaluates to `false`), Typmix *backtracks* to the latest branch node to try the next option. This branch node could be a logical disjunction, a choice between many possible computation rules, or a choice between many possible annotation rules; refer to section 2.3.1 and figure 2.5.

Because Typmix is implemented in a stateful language, backtracking requires more than a control-flow jump. Global state must be rolled back. This is done with a backtracking API, summarized in figure 2.9. At each branch, `pushcheckpoint` pushes a snapshot onto the backtracking stack. The snapshot structure, `Checkpoint`, is extensible so that extensions can save their own private state when necessary. When a clause fails, Typmix rolls back to the most recent `Checkpoint` using `backtrack`. The field `Checkpoint.restarted` counts how many times a checkpoint has been restarted via backtracking. When all options at a branch have been exhausted, Typmix uses `popcheckpoint` to discard the `Checkpoint` at that branch.

Typmix tracks four kinds of state by default: the run-time stack, which `xoc` attributes have been computed, which logical metavariables have been unified, and what error messages have been emitted. Each of these is described in turn:

Xoc’s run-time stack Typmix saves the program counter and copies the top-most stack frame. These are restored when backtracking. Note that Typmix does *not* checkpoint the entire stack, which has an important consequence: it is not possible to backtrack to a stack frame which has already been exited. This means that the *first* branch which succeeds becomes fixed, even if other branches are possible. For example, the clause $(e_1 \parallel e_2)$ will *never* try e_2 once e_1 has been found `true`. This limitation makes the implementation easy because the run-time stack correlates exactly with the depth-first search stack. Future work may eliminate this limitation by forking program state, or by re-executing the program à la Verisoft [God97].

We have not found this limitation too restrictive. The biggest problem is that we cannot implement name overloading in a nice way. (Without this restriction, it would be easy to let Typmix try all possible overloadings automatically.)

Computed attributes Typmix remembers which `xoc` attributes have been computed following a checkpoint. On backtrack, they are set to the “un-computed” state so they will be recomputed when the next branch is attempted.

Unified variables Similarly, Typmix remembers which metavariables have been unified following a checkpoint, and un-unifies each of them when backtracking.

Error messages For each branch taken from a checkpoint, Typmix saves the set of errors emitted along that branch. When a checkpoint is discarded, Typmix selects a branch and pushes all errors from that branch down the checkpoint stack. Errors are not actually printed until the last checkpoint is discarded. It is hard to know which branch to select. By default, Typmix selects the branch with the fewest number of errors, but can be configured to keep more (or even all) error messages, if desired.

2.3.5 Binary Relations

Formal type system definitions frequently use relations which are not syntax directed. The most common example is subtyping. Inference rules for such relations can be hard to evaluate because the evaluator has to guess. For example, subsumption rules require the evaluator to guess when to promote a type to a supertype. Also, transitive and reflexive closure rules require the evaluator to guess which intermediate steps complete the closure.

Because we want Typmix to be reasonably efficient, “guessing”, as described above, is not supported. All inference rules must be syntax directed. This means that subsumption must be encoded algorithmically [Pie02], and transitive and reflexive closures must be implemented by a procedural algorithm.

However, Typmix provides two things that make this task somewhat less tedious. First, much like Haskell [Jon03] and other languages, it is possible to declare that a custom infix operator should stand-in for a binary function. For example, the operator $A <: B$ can be declared to stand-in for the function `issubtype(A, B)`.

Second, Typmix provides a graph library that can be used to implement nominal binary relations, as in Java-style class subtyping. Each name is represented as a node in the graph, and directed edges between nodes denote a relation between names. The library will automatically insert edges to represent symmetric, reflexive, and transitive closures as desired. Our Java case study (Chapter 4) makes heavy use of this library.

2.4 Scoperules

As mentioned in section 2.2, the scoperules language is used to modularly and extensibly propagate the program environment. Each scope in the environment, such as the program variable bindings or the namespace of class names, is an *attribute*. Local rules define how attributes propagate among neighboring AST nodes. These rules are extensible; later rules can override or stack with earlier rules using the `default` keyword, in a way similar to `xoc`'s extensible functions. A built-in set of propagation rules *threads* attributes top-down and left-to-right through the syntax tree. This is the common case for the majority of program scopes, and eliminates many redundant hand-written rules,

but can be overridden when necessary.

Scoperules can define attributes of any possible type. Most commonly, a “scope” is a binding of names to types, so Typmix provides a generic `Scope` data structure and library for this purpose. This library can be used to express many common kinds of program scopes found in languages like ML and Java. It also provides a mechanism for extensibly specifying what sorts of bindings are legal in each scope.

This section introduces the scoperules language by example and then details its implementation.

Simply-typed lambda calculus Again, our example will be the simply-typed lambda calculus. The full language is presented in figure 1.1.

```
1 scoperules
2 {
3   vars: Scope<Simple.type>;
4
5   globals {
6     vars = new Scope();
7   }
8
9   ~expr{fun \x:\T. \e} {
10    e.vars@in = scopebind(x, T, default);
11    e.vars@out = term.vars@in;
12  }
13 }
```

Line 3 declares a new scoping attribute, `vars`. This attribute appears on every node of the AST, making scopes available from anywhere. The attribute `vars` has type `Scope<Simple.type>`, which maps names to types of type `Simple.type`. Line 3 actually declares two xoc attributes: `vars@in` and `vars@out`. These are used to thread scopes through the syntax tree: `vars@in` is the value of a scope *at* or *on* an AST node, while `vars@out` is the value of a scope *leaving* an AST node. Section 2.4.1 describes this in detail. The name `vars` as an alias for `vars@in`.²

Line 6 defines the global `vars` scope (it is empty). Lines 10 and 11 are the two propagation rules for this language. The rule on line 10 binds the variable `x` to type `T` in the scope `e.vars`. The rule on line 11 discards this binding when leaving `e`, so that the variable `x` is lexically scoped to the expression `e`. These rules use two special names, `term` and `default`. The name `term` refers to the AST node matched by the rules pattern (in this case, a function expression). The name `default` on line 10 refers to the previously-defined value of `e.vars@in` (in this case, default left-to-right threading uses the value of `T.vars@out`).

Scoperules syntax The full syntax of Typmix scoperules is given in figure 2.10.

As mentioned, the `globals` block is used to create the initial (global) value of a scope attribute. Local rules are guarded with a pattern. Each rule assigns an expression to an `@in` or `@out` attribute on a subtree of that pattern, or on the pattern itself (by using the special name `term`).

²In this thesis, we use `vars@in` inside the scoperules language and `vars` everywhere outside of the scoperules language, for example in typerules or xoc code (cf. figure 1.1).

```

Top ::= scoperules { S* }
S   ::= name : type ;
      | globals { G* }
      | pattern { R* }
G   ::= name = expr ;
R   ::= name . name@in = expr ;
      | name . name@out = expr ;

```

Figure 2.10: Syntax of Typmix scoperules

The special expression `default` is used to stack rules. For example, the following two sets of rules are equivalent:

```

~expr{fun \x:\T. \e} {
  e.vars@in = new Scope();
  e.vars@in = scopebind(x, T, default);
}

~expr{fun \x:\T. \e} {
  e.vars@in = scopebind(x, T, new Scope());
}

```

Obviously this is a pointless example; nevertheless, rule stacking with `default` is a useful way for type system extensions to inject new values into pre-existing scopes.

Advantages over xoc attributes As detailed in section 2.4.2, Typmix’s scoperules have a very direct and simple encoding in xoc attributes. So why not use xoc attributes directly? The major advantage is conciseness. A sizable amount of boilerplate code is eliminated by the default threading behavior of Typmix’s scoperules.

Further, scoperules are less fragile than xoc attributes. The grammar for the simply-typed lambda calculus might be (reasonably) refactored as below. The above scoperules will still work, but manually-coded xoc attributes would have to realize that the pattern `expr { fun \x:\T. \e }` is now two levels above `x`, not one. This is discussed further in section 2.4.2, where we give a translation from scoperules to xoc attributes.

```

grammar {
  ...
  expr: "fun" vardecl "." expr
  ...
  vardecl: var ":" type;
}

```

```

1  if(prev)
2    attr@in = prev.attr@out
3  else if(parent)
4    attr@in = parent.attr@in
5  else
6    attr@in = global
7
8  if(lastkid)
9    attr@out = lastkid.attr@in
10 else
11  attr@out = attr@in

```

Figure 2.11: Default threading rules in pseudocode

```

1  attribute
2  vars@in(term: Ast): Scope<Simple.type>
3  {
4    // default threading
7  }
8
9  attribute
10 vars@out(term: Ast): Scope<Simple.type>
11 {
12  // default threading
13 }
14
15 extend attribute
16 vars@in(term: Ast): Scope<Simple.type>
17 {
18  if(term.parent == nil)
19    return new Scope();
20  return default(term);
21 }

```

Figure 2.12: Translation of the simply-typed lambda calculus scoperules, part 1

2.4.1 Default Threading Rules

The attributes `parent` and `prev`, which are built-in to `xoc`, are used to implement top-down left-to-right threading. The attribute `parent` is `nil` for AST root nodes. The attribute `prev` is the left neighbor of an AST node, where `term` and `term.prev` both have the same `parent`. `term.prev` is `nil` when `term` is the leftmost child of its parent and when `term` has no parent. There is also an attribute `next`, which is the opposite of `prev`. It would be possible to implement right-to-left threading using `next` in place of `prev`, but we have not needed to.

Figure 2.11 shows the default threading rules in pseudocode. The `@in` attribute comes from `term.parent` when `term` is the leftmost node, from the `globals` when `term` is a root node, and from `term.prev` otherwise. The `@out` attribute comes from the rightmost child, or from the `@in` attribute when it has no children.

```

1 extend attribute
2 vars@in(term: Ast): Scope<Simple.type>
3 {
4   switch(term.parent){
5     case ~expr{fun \x:\T. \e}:
6       if(term == e){
7         _default := default(term);
8         return scopebind(x, T, _default);
9       }
10  }
11  return default(term);
12 }
13
14 extend attribute
15 vars@out(term: Ast): Scope<Simple.type>
16 {
17   switch(term.parent){
18     case ~expr{fun \x:\T. \e}:
19       if(term == e)
20         return term.vars@in;
21   }
22   return default(term);
23 }

```

Figure 2.13: Translation of the simply-typed lambda calculus scoperules, part 2

2.4.2 Translation

Scoperules are translated directly to xoc attribute functions. Figure 2.12 shows the first half of this translation for our simply-typed lambda calculus. The `vars` scope declaration is translated to the first two attribute functions in this figure. These functions implement the default threading rules for `vars@in` and `vars@out` as found in figure 2.11. The `vars` global scope rule is encoded in the third function (line 15). By definition of `globals`, this rule is active on `terms` with no parent.

Figure 2.13 shows the second half of this translation. These functions encode the rules which define function argument scoping. The encoding is straight forward, except that care must be taken to pattern match against the correct term. In this case, we pattern match against `term.parent` because the rules are defined for `e.vars`, where `e` is a subtree of the larger pattern `expr{fun \x:\T. \e}`. If the larger pattern was instead `expr{fun \x:\T. fun \x2:\T2. \e}`, the pattern match would be done against `term.parent.parent`.

2.4.3 Scope API

Typmix’s generic, reusable scope library is summarized in figure 2.14 and table 2.1. A `Scope` is an ordered set of bindings from names to types, where the binding type is a generic parameter. All operations on scopes are purely functional. This is important because the value of a scope and each at every AST node is saved as an attribute; destructive modification would be unsafe.

Name bindings are stored in `Scope.syms`, which is a list of `Sym` objects. The `Sym` object encapsulates a single name-type binding along with the location at which it was declared. (Xoc gives

```

struct Sym<T> {
    name : string
    type : T
    line : Line
}

struct Scope<T> {
    syms      : list Sym<T>
    outer     : Scope<T>
    nextframe : Scope<T>
    cache     : map (string, Sym<T>)
    depth     : int
}

typedef mergecheckfn<T>: (Sym<T>, Sym<T>) -> bool

scopebind      : (Ast, T, Scope<T>) -> Scope<T>
scopebindif    : (Ast, T, Scope<T>, mergecheckfn<T>) -> Scope<T>

scopebindlist  : (list (Ast, T), Scope<T>) -> Scope<T>
scopebindlistif : (list (Ast, T), Scope<T>, mergecheckfn<T>) -> Scope<T>

scopemerge     : (Scope<T>, Scope<T>) -> Scope<T>
scopemergeif   : (Scope<T>, Scope<T>, mergecheckfn<T>) -> Scope<T>

scopelook      : (Scope<T>, string) -> T
scopetransform : (Scope<T>, (T)->T) -> Scope<T>

```

Figure 2.14: The Scope API

us the `Line` object to represent program locations.)

The fundamental way scopes are manipulated is by *merging* two scopes; even `scopebind` is implemented by merging a scope with one element. Since scopes cannot be mutated in place, scope merging is implemented by chaining `Scope` objects, using `Scope.outer` to link to the next `Scope` in the chain. Symbols in the front of the chain shadow symbols later in the chain. This makes it easy to implement lexical scoping (the new lexical scope is simply merged in front of the outer scope).

Different scopes have different policies about how names can be shadowed. For example, ML’s `let` binding always shadows names in outer scopes, while Java has specific rules about how methods in a subclass can override methods in a superclass. These policies can be implemented by using the `if` variants of each scope merging function. Before actually merging one scope in front of another, the `if` functions first gather a list of all symbols in the “inner” scope which shadow symbols in the “outer” scope. For each shadowed symbol, the `if` functions ask a *merge check* function if it is ok to shadow one symbol with another. If the merge check fails, that new symbol is not merged in. (Merge check functions typically emit an error message explaining why the merge failed.) Since these are `xoc` functions, extensions can modify the scope merging policies of a base language by extending its merge check functions.

Typmix provides one merge check function, `scheckunique`, as part of the library. This function always fails, and is useful for scopes which require that all names be unique.

Function	Purpose
<code>scopebind(AST, T, scope)</code>	Bind a new Sym into a scope, where the sym's name is the printed string of AST and the sym's location <code>AST.line</code>
<code>scopebindlist(L, scope)</code>	Efficient shortcut for repeated calls to <code>scopebind</code>
<code>scopemerge(inner, outer)</code>	Merge the inner scope in front of the outer scope
<code>scopelook(scope, name)</code>	Lookup the type of a symbol, return <code>nil</code> if not found; the shorthand notation is <code>scope[name]</code>
<code>scopetransform(scope, F)</code>	Apply a transform function to the type of each symbol (e.g., substitute for type parameters in members of a generic class)

Table 2.1: The Scope API, explained

```

scopebindframeif : (Ast, T, Scope<T>, mergecheckfn<T>) -> Scope<T>
scopemergeframeif : (Scope<T>, Scope<T>, mergecheckfn<T>) -> Scope<T>
scopelookframe   : (Scope<T>, string) -> T

scopepushframe   : (Scope<T>) -> Scope<T>
scopeextractframe : (Scope<T>) -> Scope<T>

```

Figure 2.15: The Frames API

Frames Some languages have delimited scopes, where all symbols within the same pair of delimiters must obey some constraints. The most common example of this is block scoping of local variables in C or Java. All local variables within the same block must have a name unique to that block. The scope API described so far cannot represent this case easily. We could try using `scopebindif` to bind a local variable into a block, but this does not exactly work because it will check if the local declaration shadows any symbol, not just the symbols in the local block.

To support this common case, Typmix's scope library also supports the concept of *frames*. The API for using frames is shown in figure 2.15. First, the function `scopepushframe` merges a new (empty) scope and marks it as the end of the innermost frame. Later calls to `scopebindframeif` will ignore all shadowed symbols that are outside of the innermost frame.

Optimizations In a naive implementation, lookup operations are very inefficient. The problem is that lookup is $O(n)$ —a list of symbols must be traversed—when it should be $O(1)$. Because scope updates must be purely functional, we cannot just update a hash table, but we can cache the value of a lookup.

Typmix does this by periodically consolidating a chain of scopes into a single scope, in which all symbols are cached. The field `Scope.depth` tracks how long the chain goes to the next cache. When it reaches some (tunable) threshold, the scope is condensed and cached. (Also, consolidation can be requested explicitly when desired.)

2.5 Putting It All Together

Now we take a step back and look at how a complete type system is assembled, and at how a source file is typechecked by the xoc runtime.

A complete type system is composed of three parts: (1) the base language definition, which is one (or more) files containing the language’s grammar, typerules, scoperules, and auxiliary functions; (2) a set of type system extensions for the language, which may extend the grammar and define more typerules and scoperules; and (3) a driver script.

The driver script is the startup code executed by xoc. It is about 70 lines of boilerplate code, where about 10 of those lines need to be customized for each language. The driver script does the following:

- Import xoc and Typmix library code
- Load the base language definition
- Parse the command line
- Load extension modules specified on the command line
- Load “composition modules”
- Read, parse, and typecheck a source file

“Composition modules” are automatically loaded as dependencies of certain other module combinations. The dependencies are specified as part of the driver script. For example, a “records+subtyping” module might be loaded to add record subtyping when both “record” and “subtyping” modules are loaded separately. This is similar to the “feature dependency” capability of TinkerType. Like TinkerType, it is also possible to print an error or warning when a few of the loaded modules are known to be incompatible.

The parsing stage invokes xoc’s GLR parser, which may return an ambiguous parse. Currently in Typmix, an ambiguous parse is an error. In future work, we may try to semantically disambiguate the syntax tree automatically, as described in [Vis97].

Figures 2.16, 2.17, 2.18, and 2.19 divide the simply-typed lambda calculus as in figure 1.1 into four parts. This example is trivial, but still demonstrates the big picture. The four parts are:

- The base language (figure 2.16)
- Extension modules for nats (figure 2.17) and bools (figure 2.18)
- A composition module for “nats+bools” (figure 2.19)

If the driver script is called “simple”, it might be invoked with the command line:

```

extensible grammar Simple
{
  %ignore /[ \t\n]+/;
  %left App;
  %right Arrow;
  %priority App > Abs;

  expr: "(" expr ")"           [ paren ]
      | expr expr              [ App ]
      | "fun" var ":" type "." expr [ Abs ]
      | var;

  type: "(" type ")"         [ paren ]
      | type "->" type;     [ Arrow ]

  var: /[a-z][A-Za-z0-9_]*;/a;
}

scoperules
{
  vars: Scope<Simple.type>;

  globals {
    vars = new Scope();
  }

  ~expr{fun \x:\T. \e} {
    e.vars@in = scopebind(x, T, default);
    e.vars@out = term.vars@in;
  }
}

typerules
{
  judgement type: (term: Simple.expr) "::" Simple.type;

  ~expr{\x:var}           :: term.vars[x];
  ~expr{fun \x:\T1. \e} :: '\{\T1 -> \T2} { e :: T2 }
  ~expr{\e1 \e2}         :: T           { e1 :: {\T2 -> \T} && e2 :: T2 }
}

```

Figure 2.16: The simply-typed lambda calculus base language.

```
xoc simple -x nat -x bool [filename]
```

This chapter has motivated and present the design of Typmix in detail. The following chapters delve into more interesting case studies, demonstrating how Typmix can be used to implement and extend type systems for real languages.

```

extend grammar Simple
{
  expr: "0";
  type: "Nat";
}

scoperules
{
  globals {
    vars = scopebindlist([\var{succ}, `type{Nat->Nat}],
                          [\var{pred}, `type{Nat->Nat}]],
                        default);
  }
}

typerules
{
  ~expr{0} :: \{Nat};
}

```

Figure 2.17: “nat” extension module

```

extend grammar Simple
{
  expr: "if" expr "then" expr "else" expr [ Abs ]
      | "true"
      | "false";
  type: "Bool";
}

typerules
{
  ~expr{true} :: \{Bool};
  ~expr{false} :: \{Bool};

  ~expr{if \e then \t else \f} :: T { e :: {Bool} && t :: T && f :: T }
}

```

Figure 2.18: “bool” extension module

```

require "nat";
require "bool";

scoperules
{
  globals {
    vars = scopebind(\var{iszero}, `type{Nat->Bool}, default);
  }
}

```

Figure 2.19: “nat+bool” composition module

CHAPTER 3

Case Study: ML

This chapter demonstrates how Typmix can be used to implement and extend the type system of an ML-like functional language. We describe an implementation of the Damas-Milner type system (section 3.1) and then extend it with the following features:

- references and tuples (section 3.2)
- recursive data types and pattern matching (section 3.3)
- type qualifiers (section 3.4)
- gradual typing (section 3.5)

3.1 Damas-Milner

This section describes “Lambda”, a language that uses Damas and Milner’s polymorphic type system [DM82]. Figure 3.1 shows the core grammar, typerules, and scoperules for Lambda. The grammar and rules for this language are very similar to those in figure 1.1, which defines a simply-typed language. The important difference is under the covers: Lambda extends Typmix’s default unification algorithm to infer the types of type variables (`typevars`) and to instantiate polymorphic (`forall`) types. The following describes how this is done.

In line 18 each function parameter is bound to a fresh type variable (a `typevar`). Typmix needs to know that `typevars` are metavariables which can be unified. As described in section 2.3.2, this is what the `logicalvar` attribute is for—every AST node that implements this attribute (by returning a non-null `LogicalVar` object) is considered a metavariable. Therefore, Lambda extends this attribute so that `typevars` are unified like other metavariables. The code for doing so looks like the following:

```
extend attribute
logicalvar(term: Ast): LogicalVar
{
  if(term ~ type{\_::typevar})
    return new LogicalVar();
  return default(term);
}
```

Lines 25-26 generalize the inferred type of `e1` to a polymorphic type. On line 25, `cangeneralize` performs the value check [Wri95]. On line 26, `generalize` collects all free type variables in the

```

1  extensible grammar Lambda
2  {
3    expr: expr expr
4        | "fun" var "." expr
5        | "let" var "=" expr "in" expr
6        | var;
7
8    type: type "->" type
9         | "forall" typevar "." type
10        | typevar;
11 }
12
13 scoperules
14 {
15   vars: Scope<Lambda.type>;
16
17   ~expr{fun \x. \e} {
18     x.vars@in = scopebind(x, mktypevar(term.letrank), default);
19     e.vars@out = term.vars@in;
20   }
21
22   ~expr{let \x = \e1 in \e2} {
23     e.vars@in = (if cangeneralize(e1) then
24                 scopebind(x, generalize(e1.type, term.letrank), default)
25                 else
26                 scopebind(x, e1.type, default));
27     e.vars@out = term.vars@in;
28   }
29 }
30
31 typerules
32 {
33   judgement type: (term: Lambda.var) "::<" Lambda.type;
34   judgement type: (term: Lambda.expr) "::<" Lambda.type;
35
36   ~var{\x}                :: term.vars[x];
37   ~expr{\x::var}          :: x.type;
38   ~expr{fun \x. \e}       :: \{\T1 -> \T2} { x :: T1 && e :: T2 }
39   ~expr{let \x = \e1 in \e2} :: T2      { e1 :: T1 && e2 :: T2 }
40   ~expr{\e1 \e2}         :: T          { e1 :: {\T2 -> \T} && e2 :: T2 }
41 }

```

Figure 3.1: Grammar, scoperules, and typerules for Lambda.

inferred type and “generalizes” them by constructing a polymorphic type. Each `typevar` is annotated with its `letrank`, which is used to determine if a `typevar` is bound in the environment; such `typevars` can not be generalized. The attribute `letrank` (not shown) counts the nesting depth of `let` expressions, as described in [Rem92].

Finally, Lambda extends the unification algorithm in two ways. First, recall that the default unification algorithm does not do an occurs check (section 2.3.2). Lambda adds an occurs check for type variables by extending `unifyvar` like this:

```
extend fn
unifyvar(var: LogicalVar, term: Ast, state: UnifyState): bool
{
  if(term ~ type && occurs(var, term))
    return unifyerror("occurs check failed");
  return default(term);
}
```

Next, Lambda extends `unifystruct` to support polymorphism: each polymorphic (`forall`) type is instantiated with a fresh type variable before being unified. The following outline demonstrates how this is done:

```
extend fn
unifystruct(A: Ast, B: Ast, state: UnifyState): bool
{
  if(A ~ type{forall \x. \T}){
    xnew := mktypevar();
    return unify(typesubst(T, x, xnew), B, state);
  }
  ...
  return default(A, B, state);
}
```

The net effect of these changes makes `unify` act exactly like algorithm J [Mil78]. Of course, none of these algorithms are new, but that is not important. What is important is that their implementation in Typmix is *extensible* and *generic*. The extensibility comes from `xoc`’s extensible functions. Thanks to extensible functions, we were able to implement Damas-Milner style unification as a modular extension of Typmix’s default algorithm. Later extensions will further extend the algorithm presented in this section by extending `unifyvar` and `unifystruct` in similar ways. By generic, we mean that these algorithms, specifically `unify` and `generalize`, ignore parts of the syntax tree that they do not care about. This is possible thanks to `xoc`’s library of generic traversals, and specifically `astsplit` (section 2.1). As a consequence, these algorithms will still work even when other Typmix extensions add new syntax. For example, the two extensions presented in the next section add new syntax but do not need to extend `generalize` or `unify`.

3.2 References and Tuples

Adding reference types, as in figure 3.2, is almost embarrassingly easy. Not shown is the imported “seq” extension (of similar size) which adds sequences (`e1 ; e2`) and the unit type `()`.

```

1 require "seq";
2
3 extend grammar Lambda
4 {
5     expr: "ref" expr
6         | "!" expr
7         | expr "!=" expr;
8
9     type: "Ref" type;
10 }
11
12 typerules
13 {
14     ~expr{ref \e}      :: `{\Ref \T} { e :: T }
15     ~expr{! \e}       :: T          { e :: {\Ref \T} }
16     ~expr{\e1 := \e2} :: `{()}      { e1 :: {\Ref \T} && e2 :: T }
17 }

```

Figure 3.2: References extension

```

1 extend grammar Lambda
2 {
3     expr: "(" expr "," expr[,+]"
4         | expr "." tupleproj;
5     tupleproj: /[0-9]+/;
6
7     type: "(" type "," type[,+]" ";
8 }
9
10 typerules
11 {
12     judgement types: (terms: Lambda.expr[,+]" ::" Lambda.type[,+];
13
14     ~expr[,+] :: listmap(terms, fn(e){ return e.type })
15         { forall(e in terms) {e :: _} }
16
17     ~expr{(\e, \rest)} :: `{\T, \Trest}          { e :: T && rest :: Trest }
18     ~expr{\e . \nth}   :: projtuple(nth, T # Trest) { e :: {\T, \Trest} }
19 }
20
21 fn
22 projtuple(n: Lambda(tupleproj), tuple: list Lambda.type): Lambda.type
23 {
24     nth := toint(n);
25     if(nth >= length(tuple)){
26         error("out-of-range tuple projection");
27         return nil;
28     }
29     return tuple[nth];
30 }

```

Figure 3.3: Tuples extension

```

1  extend grammar Lambda
2  {
3    expr: "data" datavar typevar* "=" datacons[|]+ "in" expr
4        | "type" typevar typevar* "=" type "in" expr
5        | consvar expr
6        | "case" expr "of" caseexpr[|]+;
7
8    caseexpr: pattern "->" expr;
9    pattern: consvar pattern
10         | patvar;
11    datacons: consvar type;
12
13    type: "fun" typevar "." type
14        | type type
15        | datavar;
16
17    kind: "*" | kind "->" kind;
18 }

```

Figure 3.4: Grammar extensions for Lambda type constructors and pattern matching

The tuples, shown in figure 3.3, is of similar size. The tuple grammar uses list symbols like `expr [,]+`, which is a shorthand for a list of comma-separated expressions with at least one element. Note that grammar lists can be used in syntax patterns and unified like other ASTs, but can also be “consed” and iterated over like other lists. Lines 12-15 define a typing judgement for `expr [,]+` lists: the list of expressions is mapped to a list of types. The function `projtuple` on line 15 returns the n th element of the list $T \# Trest$ (where $\#$ is “cons”) or returns an error if out-of-range.

Also note that the tuple syntax cannot be simplified to `" (" expr [,]+ ") "` because that would be ambiguous with `" (" expr ") "`. A language designed with tuples from the start would not have this problem. This simple example demonstrates the inherent conflict between extensibility and simplicity. More examples are encountered later. The key consequence is that it is sometimes “uglier” to implement a feature as an extension, rather than as a built in. Nevertheless, the fact that such features *can* exist as extensions is vital. The goal of Typmix, after all, is to enable the creation of pluggable and extensible type systems.

3.3 Data Types and Pattern Matching

This section describes an extension that adds data types and pattern matching to Lambda. This extension actually implements constructor classes [Jon93], which are a more general notion of data types that require a system of kinding and kind inference. Typmix’s typerules can be used to naturally implement the required kinding system, demonstrating how they are useful for more than just expression typing. The new syntax is summarized in figure 3.4. The following examines interesting aspects of this extension.

Kinding We extended the grammar of types to include type abstraction and application. The typerules for this system, shown in figure 3.5, are a straightforward encoding of the usual kinding rules

```

1 typerules
2 {
3   judgement kind: (type: Lambda.type) "::" Lambda.kind;
4
5   ~type{\x}           :: type.typevarkinds[x];
6   ~type{\A -> \B}    :: '{*}           { A :: {*} && B :: {*} }
7   ~type{forall \x. \T} :: '{*}           { T :: {*} }
8   ~type{fun \x. \T}   :: '{\K1 -> \K2} { x :: K1 && T :: K2 }
9   ~type{\A \B}       :: K             { A :: {\K2 -> \K} && B :: K2 }
10 }

```

Figure 3.5: Kinding rules

as found in [Pie02] or [Jon93]. The interesting aspect of our kinding extension is how it affects type unification. Following [Jon93], we extend `unify`¹ to ensure that `typevars` are only unified with types of the same kind. Further, we extend `unify` to rewrite type applications into a canonical form.

Type constructors A `type` expression declares a non-recursive type constructor, and a `data` expression declares a recursive data type constructor. Below is an (admittedly silly!) example of their use. The declaration `List` adds a new type of form `(fun a. List a)` which has kind `(*->*)`. It also adds two new functions, whose types are shown below in comments.

```

data List a = Nil () | Cons (a, List a)
type ApplyNat c = c Nat

// Nil  :: forall a. () -> List a
// Cons :: forall a (a, List a) -> List a

```

Pattern matching Pattern matching is very easy to implement, as shown by the `typerules` and `scope-rules` in figure 3.6. It is also easy to extend the sorts of types that can be pattern matched. Figure 3.7 shows an extension that adds tuple pattern support. (Not shown is the judgement which types a list of patterns.) This “composition extension” might be loaded automatically when both the tuple and core pattern matching extensions are loaded, as described in section 2.5.

3.4 Type Qualifiers

This section shows how to implement a framework for type qualifiers [FFA99] on top of `Lambda`. Most interestingly, this framework supports qualifier inference and qualifier polymorphism.

The grammar in figure 3.8 adds qualified types, along with qualifier annotations and assertions. Line 9 of the grammar declares a negation operator for qualifiers. For example, if `const` is a “constant” qualifier, then `~const` means “not constant”. Because it is meant to be a general framework,

¹For brevity, the rest of this thesis uses `unify` to collectively refer the more specific `unifyvar` and `unifystruct` functions, described earlier.

```

1 scoperules
2 {
3   patvars: Scope<Lambda.type>;
4
5   ~caseexpr{\p -> \e} {
6     p.patvars@in = new Scope();
7     e.patvars@in = term.patvars@in;
8     e.vars@in = scopemerge(p.patvars@out, default);
9   }
10
11   ~patvar{\x} {
12     term.patvars@in = scopebindif(x, mktypevar(), default, scheckunique);
13   }
14 }
15 typerules
16 {
17   judgement type: (term: Lambda.pattern) "::" Lambda.type;
18
19   ~pattern{\x}      :: type.patvars[x];
20   ~pattern{\cons \p} :: Tdata      { cons :: {\T -> \Tdata} && p :: T }
21
22   ~expr{case \root of \cases} :: T {
23     root :: Troot &&
24     forall(~{\p -> \e} in cases) {
25       p :: Troot && e :: T
26     }
27   }
28 }

```

Figure 3.6: Typerules and scoperules for pattern matching

```

1 require "tuple";
2 require "pattern";
3
4 extend grammar Lambda
5 {
6   pattern: "(" pattern "," pattern[,"+ "]" ;
7 }
8
9 typerules
10 {
11   ~pattern{(\first, \rest)} :: '{(\T, \Trest)} { first :: T && rest :: Trest }
12 }

```

Figure 3.7: Extending pattern matching with tuple support

```

1 extend grammar Lambda
2 {
3   expr: "assert" "(" expr "," qual+ ")"
4       | "(" qual ")" expr;
5
6   type: qual type
7       | "forall" qualvar "." type;
8
9   qual: "~" qual;
10 }

```

Figure 3.8: Grammar extension for type qualifiers

the grammar in figure 3.8 does not define any actual qualifiers. Instead, qualifiers are implemented as further extensions on top of this one. Each extension that adds a new qualifier only does two things: declare whether the new qualifier is “positive” or “negative” (as defined in [FFA99]) by extending the `qualsign` function, and declare typing rules that define semantics for the new qualifier. Everything else is handled automatically by the framework, as described in the following.

Qualifiers form a lattice, which is used to define a subtyping relation. For example, under the usual semantics of `const` we expect `T <: const T` to hold for all `T`. Inversely, we expect that `~const T <: T`. We implement the subtyping relation for this extension using a function named `issubtype`. Because `Typmix` does not support arbitrary subsumption rules (section 2.3), subtyping must be encoded algorithmically. For example, we add another rule for function application:

```
typerules {
  infix issubtype: "<:";

  ~expr{\e1 \e2} :: T { e1 :: {\T2 -> \T} && e2 :: T22 && T22 <: T2 }
}
```

Importantly, `issubtype` must infer qualifier bounds in order to support qualifier inference. Qualifier inference differs from basic Lambda type inference in two ways. First, a qualifier variable actually unifies with a *set* of qualifiers, whereas a type variable unifies with a single type. Second, the order that qualifiers appear on a type is irrelevant. Both of these points imply that the inference algorithm must look at all of a type’s qualifiers at once. Additionally, we need to extend `LogicalVar` to track a variable’s set of bound qualifiers. This is summarized in the following pseudocode:

```
extend struct LogicalVar {
  unifiedquals: list Lambda.qual;
}

extend fn
issubtype(A: Lambda.type, B: Lambda.type) {
  if(A.qualifiers || B.qualifiers){
    if(!issublattice(A.qualifiers, B.qualifiers))
      return false;
    // ... if A or B contain qualifier vars, unify them here ...
    return issubtype(A.stripped, B.stripped);
  }
  return default(A, B);
}
```

Next we need to update typing rules to create qualifier variables in the appropriate places. For example, the following snippet shows a qualifier-aware typing rule for reference constructors. This is exactly the rule in figure 3.2, augmented with a qualifier variable on the result type. Foster et al. describe a spread function `sp` which inserts qualifier variables where necessary. Ideally our qualifier framework would apply `sp` automatically but it currently does not, which is disappointing.

```
typerules {
  ~expr{ref \e} :: `{\(mkqualvar()) Ref \T} { e :: T }
}
```

```

1 require "int";
2 require "quals";
3
4 extend grammar Lambda
5 {
6   qual: "pos" | "neg" | "zero" | "nonzero";
7 }
8 typerules
9 {
10  ~expr{0}           :: ` {zero \T}  { term::T }
11  ~expr{\e1 + \e2}  :: ` {pos \T}   { term::T && e1:: {pos \_} && e2:: {pos \_} }
12  ...
13  ~expr{\e1 * \e2}  :: ` {pos \T}   { term::T && e1:: {neg \_} && e2:: {neg \_} }
14  ...
15  restrict
16  ~expr{\e2 / \e2}  :: T {
17    where(e2 :: {zero \_})
18    warn("division by zero")
19  }
20 }
21 extend fn
22 qualsign(Q: Lambda.qual)
23 {
24   if(Q ~ {pos} || Q ~ {neg} || Q ~ {nonzero} || Q ~ {zero})
25     return QualNegative;
26   return default(Q);
27 }

```

Figure 3.9: Integer qualifiers

Finally, the framework extends `generalize` to universally quantify all free qualifier variables in a type, and extends `unify` to instantiate polymorphic qualifier types. These straightforward changes are used to support qualifier polymorphism.

We have implemented two qualifiers extensions using this framework: integer qualifiers (figure 3.9) and a `const` qualifier for references (Figure 3.10).

3.5 Gradual Typing

Gradual typing [ST06] was proposed as one way to unify static and dynamic typing. It introduces a new type, `?`, which is the dynamic type, and defines a consistency relation which allows static types to be converted to the dynamic type (we use $=\sim$ for the consistency relation). The presentation in [ST06] does not include an inference algorithm, so it does not naturally fit into Lambda. Consequently, the extension presented here actually extends Simple (from figure 1.1), not Lambda. The extension is shown in figure 3.11, and is straightforward. The `consistentypes` function simply does a case analysis on the structure of each type to implement the rules defined in [ST06].

```

1 require "ref";
2 require "quals";
3
4 extend grammar Lambda
5 {
6   qual: "const";
7 }
8 typerules
9 {
10  restrict
11  ~expr{\e1 := \e2} :: _ {
12    e2 :: {~const Ref \_} || error("assignment to const reference")
13  }
14 }
15 }
16 extend fn
17 qualsign(Q: Lambda.qual)
18 {
19   if(Q ~ {const})
20     return QualPositive;
21   return default(Q);
22 }

```

Figure 3.10: const reference qualifier

```

1 require "ref";
2
3 extend grammar Simple
4 {
5   type: "?";
6 }
7
8 fn consistenttypes(A: Simple.type, B: Simple.type): bool {
9   ...
10 }
11
12 typerules
13 {
14   infix consistenttypes: "=~";
15
16   ~expr{\e1 \e2}   :: '{?}'   { e1 :: {?} && e2 :: _ }
17   ~expr{\e1 \e2}   :: T       { e1 :: {\T2 ->\T} && e2 :: S2 && T2 =~ S2 }
18   ~expr{! \e}      :: '{?}'   { e :: {?} }
19   ~expr{\e1 := \e2} :: '{()}'  { e1 :: {?} && e2 :: _ }
20   ~expr{\e1 := \e2} :: '{()}'  { e1 :: {Ref \T} && e2 :: S && T =~ S }
21 }
22 }

```

Figure 3.11: Gradual typing extension

CHAPTER 4

Case Study: Java

This chapter demonstrates how Typmix can be used to implement and extend type systems for Java-like imperative, object oriented languages. Java presents a different set of challenges for type system implementors. For one, even the minimal subset of Java presented here is much larger than ML. Whereas most of the ML language is focused almost purely on typing and unification, much of the work needed to implement Java revolves around a more complicated set of scoping rules. Further, the imperative nature of Java gives us an opportunity to experiment with type-and-effect systems (such as the last two extensions below).

We describe an implementation of a subset of Java which very closely matches Featherweight Java (section 4.1), and then extend it with the following features:

- standard primitive types and control-flow structures (section 4.2)
- generics (section 4.3)
- checked exceptions (section 4.4)
- type-based race detection (section 4.5)

Two of these extensions stand out for their potential real-world value. First, generics were recently added to Java 1.5 [GJS05], so they represent a “real” Java extension. Second, as mentioned earlier, checked exceptions are a somewhat controversial language feature [Hej03]. This makes them a good example of an extension that should be “pluggable” but isn’t.

4.1 Featherweight Java

In this section we discuss an implementation of Featherweight Java [IPW01] in Typmix. The language we describe is actually more expressive than Featherweight Java, but its minimalness makes it similar in spirit.

Grammar A fragment of the grammar is shown in figure 4.1. Two things are interesting about this grammar. First, it is wordy; declarations are split into many fine grained non-terminals. This is a result of conscious planning ahead. When syntax is not divided up into a fine grained way, extensions become burdensome to write. For example, consider writing the checked exceptions extension (section 4.4). If `method` and `proto` were collapsed, this extension would add a new rule like:

```

1 extensible grammar Java
2 {
3   top: classdef*;
4
5   classdef: classheader classbody;
6   classheader: "class" classname classextends;
7   classextends: "extends" userclasstype;
8   classbody: "{" field* constructor method* "}";
9
10  constructor: classname "(" fnarg[,]* ")" block;
11  field: usertype name ";"
12  field: usertype name ";";
13
14  stmt: ...
15  expr: ...
16
17  usertype:      userclasstype;
18  userclasstype: classname;
19
20  type: <Class>  classtype
21       | <Method> methodtype
22       | <Null>  "null";
23
24  methodtype: type "(" type[,]* ")";
25  classtype: "class" classname;
26 }

```

Figure 4.1: Java grammar

```
method: usertype name "(" fnarg[,]* ")" "throws" userclasstype[,]+ body;
```

The extension would then have to duplicate all existing scoping rules for methods to accommodate this new syntax. Dividing `method` into `method` and `proto` greatly improves modularity. Another notable example is `classdef`, which was split into four pieces to accommodate generics (section 4.3). The need to plan ahead to maximize modularity is somewhat disappointing because it puts an extra burden on Typmix developers. However, as mentioned in section 3.2 (where we encountered a similar problem), it is important to note that Typmix makes such extensibility possible in the first place.

The second interesting point about this grammar is the division between `usertype` and `type`. This is necessary because some Java types appear internally but not externally; namely, method types and the special `null` (bottom) type. This requires a translation of `usertypes` to `type`, which is explained next.

Typing Figure 4.2 summarizes the typing judgements used to typecheck Java programs. They are divided into four groups. The first group (lines 6-7) converts a `usertype` to an internal `type`, using “=>” instead of “: :” to emphasize that it represents a conversion. The second group (lines 9-10) extracts a type from a declaration. The third group (lines 12-13) types expressions and lists of expressions. The fourth and final group (lines 15-23) checks that a statement or declaration is “well typed”, or free of typing errors. A sampling of the actual typing rules is shown in figure 4.3. For the “well typed” rules, the name `OK` is an alias for the constant `true`.

```

1 typerules
2 {
3   infix issubtype: "<:";
4   infix aresubtypes: "<:";
5
6   judgement type:      (term: Java.usertype)      "=>" Java.type;
7   judgement classtype: (term: Java.userclasstype) "=>" Java.type.Class;
8
9   judgement type: (term: Java.fnarg) "=>" Java.type;
10  judgement type: (term: Java.proto) "=>" Java.type;
11
12  judgement type: (term: Java.expr) "::" Java.type;
13  judgement types: (terms: Java.expr[,]*) "::" Java.type[,]*;
14
15  judgement welltyped: (term: Java.classdef)      "::" bool;
16  judgement welltyped: (term: Java.classheader)  "::" bool;
17  judgement welltyped: (term: Java.classextends) "::" bool;
18  judgement welltyped: (term: Java.classbody)    "::" bool;
19  judgement welltyped: (term: Java.constructor)  "::" bool;
20  judgement welltyped: (term: Java.field)        "::" bool;
21  judgement welltyped: (term: Java.method)       "::" bool;
22  judgement welltyped: (term: Java.block)        "::" bool;
23  judgement welltyped: (term: Java.stmt)         "::" bool;
24 }

```

Figure 4.2: Java typing judgements

```

1 ~userclasstype{\c::classname} => term.classtable[c];
2 ...
3
4 ~fnarg{\t \name} => T { t => T }
5 ...
6
7 ~expr{this}          :: term.thistype;
8 ~expr{(\t) \e}      :: T { t => T && e :: A && (A <: T || T <: A) }
9 ~expr{new \c (\args)} :: C {
10   c :: C && args :: Targs &&
11   C.members["<init>"] ~ {\T (\Cargs)} && Targs <: Cargs
12 }
13 ...
14
15 ~classbody{\fields \cons \methods} :: OK {
16   forall(f in fields) { f :: OK } &&
17   forall(m in methods) { m :: OK } &&
18   cons :: OK
19 }
20 ...
21
22 ~stmt{return \e;} :: OK { e :: T && T <: term.returntype }
23 ...

```

Figure 4.3: A sample of Java typing rules

```

1  scoperules
2  {
3      classtable:  Scope<Java.type.Class>;
4      classmembers: Scope<Java.type>;
5
6      thistype:   Java.type.Class;
7      supertype:  Java.type;
8      returntype: Java.type;
9
10     vars: Scope<Java.type>;
11 }
12
13 attribute Java.type.Class :: members : Scope<Java.type>;

```

Figure 4.4: Java scopes

Line 11 types constructor calls. The attribute `members` gives the members scope of a class type, where “<init>” is the special name for the constructor’s method type.

Subtyping is implemented using Typmix’s graph library for nominal relations (section 2.3.5). The function `issubtype` checks if one type is subtype of another by querying the graph library. It also has a hard-coded check for `null`, the bottom type. The function `aresubtypes` checks if one list of types is a subtype of another. It is used in line 11 of figure 4.3 to check subtyping of constructor arguments. Lastly, the function `jointypes` computes the least-upper bound of two types.

Scoping Even a subset of Java as minimal as this one has a variety of scopes. They are summarized in figure 4.4. The `classtable` scope maps names to class types, and the `vars` scope maps local variable bindings. The three attributes on lines 6-8 track certain special types (e.g., `thistype` and `supertype` are the type of `this` and `super`, respectively). Note that `super` is a method type in constructors, while `this` is always a class type.

Figure 4.5 gives a sample set of Java scoping rules. The most interesting scopes are `classmembers` and `members`. The scope `classmembers` is used to gather member declarations inside a class body, and is not directly referenced by any typerules. Java’s `Object` type is a built-in member of this scope. The scope `members` is an attribute on `Java.type.Class` ASTs. It contains all the members of a class type. This is the scope used by typerules, as in line 11 of figure 4.3. It is necessary to have both `classmembers` and `members` because *all* methods of a class are visible in every other method. So `classmembers` is essentially a temporary scope used to construct `members`. The rule for binding fields into the `classmembers` scope (line 11 of figure 4.5) demonstrates how Typmix’s scoperules make it easy to construct scopes without ever having to write a traversal function.

The ugliest part of all of this is the implementation of the `members` attribute. On the surface, the `members` attribute can be easily implemented with the following definition, where `super` is `C`’s superclass:

```

attribute
members(C: Java.type.Class): Scope<Java.type>
{
    return scopemergeif(classbody.classmembers@out, super.members, scheckoverride);
}

```

```

1  scoperules
2  {
3      ~classdef{class \name \extends \body} {
4          name.classtable@in = scopebind(name, mkclasstype(term), default);
5          body.classtable@in = new Scope();
6          term.classtable@out = term.classmembers@in;
7      }
8      ...
9
10     ~field{\t \name ;} {
11         term.classmembers@out = scopebindif(name, t.type, default, scheckunique);
12     }
13     ...
14
15     ~method{\proto \body} {
16         proto.vars@in = scopepushframe(term.thistype.members);
17         term.vars@out = term.vars@in;
18     }
19
20     ~fnarg{\t \name} {
21         term.vars@out = scopebindframeif(name, t.type, default, scheckunique);
22     }
23     ...
24 }

```

Figure 4.5: Some of Java’s scoping rules

```

}

```

But where does `classbody` come from? And where does `super` come from? The answer is that we have to extend `xoc`’s `Ast` structure with a field that is set by `mkclasstype`. This field, accessible as `C.classdef`, points to the AST at which the class was defined. From that AST we can extract `classbody` and the `super` type. This is the only time we had directly muck with `xoc`’s `Ast` structure, but we still feel that it is unfortunate and would prefer a solution which uses only the more common mechanisms (typerules, scoperules, and attributes). The following shows how `Ast` is extended:

```

extend struct Ast.type.Class {
    classdef: Java.classdef;
}

```

The usefulness of Typmix’s scope API is demonstrated nicely in figure 4.5. Line 11 binds a new field, which must have a unique name. This is checked automatically by the merge check function `scheckunique`. Line 16 imports the entire `members` scope of the current class into the method’s local variables. Later, the local variables will be bound directly into this scope. (And they are bound using frames, so that names clashes within each local block can be detected.) By merging the `members` scope with the `vars` scope, Typmix has concisely encoded the following rule, which other compilers encode with a procedural algorithm: “To look up a variable in a method body, first look in the locally declared set of variables, and then look in the class members.”

```

1  extend grammar J̄ava
2  {
3      stmt: block
4          | usertype name ";"
5          | "if" "(" expr ")" stmt "else" stmt
6          | "while" "(" expr ")" stmt
7          | "break" ";" | "continue" ";"
8          | "try" stmt catch* finally?
9          | "throw" expr ";;"
10
11     catch:  "catch" "(" usertype name ")" stmt;
12     finally: "finally" stmt;
13
14     expr:  expr "||" expr
15          | expr "+" expr
16          | ...
17          | expr "[" expr "]"
18          | "new" usertype "[" "]"
19          | "true" | "false"
20          | numberchar | numberint | numberfloat;
21
22     usertype: primtype | usertype "[" "]" ;
23     primtype: "void" | "bool" | "char" | "int" | "float";
24
25     type: primtype
26         | type "[" "]" ;
27 }

```

Figure 4.6: Grammar of primitive types and control structures

4.2 Primitive Types and Control Structures

We built an extension to the minimal Java-like language in the previous section called “std”, which adds some standard primitive types, expressions, and control flow structures. In this section, we briefly discuss its interesting aspects. Figure 4.6 shows its grammar, and figure 4.7 shows a sample of its typing and scoping rules.

The first typerule (line 3) shows how to type overloaded number operators. Here, we’ve extended `issubtype` to implement a subtyping relation over numbers, and `jointypes` to “promote” two numbers to their least supertype. The second typerule (line 6) works in tandem with the scoperule at line 25 to verify that all `break` statements have a target to break to; `continue` is analogous. The remaining scoperules show how to manage local variable declarations: a frame is pushed and popped at each block and new variables are bound to the innermost frame.

The last clause of the typerule at line 10 of figure 4.7 may seem curious. It is needed because `vars` is expanded lazily—if a set of statements do not require the `vars` scope, it is not computed. For example, without that clause the error in the following block would not be caught since the `return` statement does not inspect `vars`:

```

{
  int x;
  ...
  int x; // invalid !
  return 0;
}

```

```

1 typerules
2 {
3   ~expr{a + \b} :: jointypes(A,B) { a::A && b::B && isnum(A) && isnum(B) }
4
5
6   ~stmt{break ;} :: OK {
7     term.breakstack != nil || error("illegal break");
8   }
9
10  ~block{{\stmts}} :: OK { forall(s in stmts) {s::OK} && stmts.vars@out != nil }
11 }
12 scoperules
13 {
14   breakstack: list Java.stmt;
15
16   ~stmt{\b::block} {
17     b.vars@in = scopepushframe(default);
18     term.vars@out = term.vars@in;
19   }
20   ~stmt{\t \name ;} {
21     term.vars@out = scopebindframeif(name, t.type, default, scheckunique);
22   }
23   ~stmt{while(\e) \body} {
24     body.breakstack@in = term # default;
25     term.breakstack@out = term.breakstack@in;
26   }
27 }

```

Figure 4.7: Sample typerules and scoperules for primitive types and control structures

}

4.3 Generics

In this section we describe an extension that implements generics as in Java 1.5 [BOS98]. The grammar for this extension is shown in figure 4.8.

The new grammar rule at line 10 represents both a “raw” polymorphic class type and a “substituted” class type. The `classtable` binds names to “raw” types, such as `List<X>`. Substituted versions of the raw type are obtained by substituting for each type parameter, as in `List<Integer>`. In order to tell the difference between raw and substituted class types, we unfortunately have to muck with the `Ast` structure again, as shown below. Importantly, the `members` attribute is extended in order to transform the type of all class members for substituted types. Pseudocode for this is shown below.

```

extend struct Ast.type.Class {
  issubstituted: bool;
}
extend attribute
members(C: Java.type.Class): Scope<Java.type>
{
  if(C.issubstituted)
    return scopetransform(default(C), substclassparams);
  else

```

```

1 extend grammar Java
2 {
3   classheader: "class" classname "<" usertypeparam[,+ ">" classtextends;
4   method: "<" usertypeparam[,+ ">" method;
5
6   usertypeparam: classname
7                 | classname "extends" userclasstype;
8   userclasstype: classname "<" usertype[,+ ">";
9
10  classtype: <Poly> "class" classname "<" type[,+ ">"
11             | <Param> classname;
12
13  methodtype: "<" type[,+ ">" type "(" type[,* " )";
14 }

```

Figure 4.8: Grammar extensions for Java generics

```

1 ~classdef{class \name \params \extends \body} {
2   p.classtable@in      = term.classtable@in;
3   name.classtable@in  = scopebind(name, mkclasstype(term), p.classtable@out);
4   extends.classtable@in = name.classtable@out;
5   // don't keep class params
6   term.classtable@out  = scopebind(name, name.classtype, p.classtable@out);
7 }
8 ~usertypeparam{\name} || ~usertypeparam{\name extends \_} {
9   term.classtable@in
10    = scopebindif(name, mkclassparam(term), default, scheckunique);
11 }

```

Figure 4.9: Scoping rules for polymorphic class parameters

```

    return default(C);
}

```

Each parameter for polymorphic classes is added to the `classtable` and the subtyping graph (where a parameter's supertype is its upper bound). The scoperules for these parameters are shown in figure 4.9. The first four rules specify a contorted threading of the `classtable`. The reason for this contortion is that the class type (line 3) cannot be computed until each of its parameters has been bound in the `classtable`. Thus, the `classtable` must be first threaded through the class' parameters.

Finally, we describe polymorphic method invocation. The substitutions for all type parameters at a polymorphic call must be inferred. The typerule for doing this is shown below. It relies on the function `infercalltype`, which implements the following algorithms (as described in [BOS98]): A fresh type variable is substituted for each type parameter in `Tformals`. Next `Tformals` is unified with `args.types` by calling a modified version of `unify` that computes the most specific type of each type variable while respecting the upper bound of each parameter. On success, the unified type variables are substituted into `Treturn`.

```

~expr{\f(\args)} :: infercalltype(args, Treturn, Tformals, params) {
  f :: {<\params> \Treturn (\Tformas)}
}

```

4.4 Checked Exceptions

Checked exceptions are an example of a “type-and-effect” system. In this case, the “effect” is a thrown or caught exception; the type checker verifies that all thrown exceptions are eventually caught by the program. (Java does not check exceptions that are subtypes of `RuntimeException`; we ignore this point in our extension.) In order to verify this, the type checker tracks the set of possibly-caught exceptions at every program point.

Annotations Each method is annotated with the set of exceptions it may throw. This presents a small problem: how do we store and later extract this annotation from a method type? A compiler written in a traditional language (like Java) would simply add a new field to the class that represents method types. However, in Typmix we represent method types (and all other types!) with ASTs. The naive solution is to add a new `type` grammar rule, as in:

```
type: type "(" type[,]* ")" "throws" type[,]*;
```

This is not modular: the existing rules for a method call must be updated (i.e., copied) to support this new syntax. A better solution is to annotate the type. Borrowing syntax from Java 1.5 metadata annotations, it might look like:

```
type: "@" "(" annotation ")" type;  
annotation: "throws" type[,]*;
```

Existing rules will still not work; a method may have a type that looks like `@(throws IOException) void (void)`, which does not match the type pattern `{\Treturn (\Targs)}` used by method call typing rules. Since we can extend Typmix’s unification algorithm, this is easy to fix. We extend `unify` to ignore annotations in the type that do not appear in the pattern, so that the above pattern will correctly match the above type, and all existing rules for method calls will continue to work. Further, we re-order annotations that do appear in the pattern so that the order of annotations in the type does not matter. Because our next extension (type-based race detection) uses a similar style of annotation, we wrapped this capability into a reusable module. (This module could be extended to support full Java 1.5 style metadata, but we have not done so; we only use it as an internal representation for type annotations.)

Typmix makes extensive use of ASTs, and the above example demonstrates both good and bad effects of this reality. On the bad side, using ASTs as the core compiler data structure can lead to slightly awkward representations. However, by making pattern unification extensible, Typmix allows extension writers to keep a more pleasant and abstract view of the actual concrete representation.

Checked Exceptions The important syntax, scoperules, and typerules for checked exceptions are shown in figure 4.10. This extension is very straightforward. The scoperules propagate the “caught” set, and the typerules enforce it. Not shown are the rules to translate the method prototype declaration AST to an internal `type` AST. Also, on line 15 we elide the fact the `ex` must be converted from a `userclasstype[,]+` to the internal `type[,]+`.

```

1  require "std";
2  require "annotate";
3
4  extend grammar Java
5  {
6    proto: proto "throws" userclasstype[,,+];
7    annotation: "throws" type[,,+];
8  }
9
10 scoperules
11 {
12   caught: list Java.type;
13
14   ~proto{\proto throws \ex} {
15     term.caught@out = listconcat(ex, default);
16   }
17   ~method{\proto \body} {
18     term.caught@out = term.caught@in;
19   }
20
21   ~stmt{try \s \catches \finally} {
22     s.caught@in = listconcat(gathertypes(catches), default);
23     s.caught@out = term.caught@in;
24   }
25 }
26
27 typerules
28 {
29   restrict
30   ~expr{\m(\args)} :: _ {
31     where(m :: {@(throws \ex) \M}){
32       forall(T in ex){
33         exists(C in term.caught)
34           {T <: C}
35         || error(string(C) + " isn't caught")
36       }
37     }
38   }
39
40   restrict
41   ~stmt{throw \e;} :: _ {
42     where(e :: T){
43       exists(C in term.caught)
44         {T <: C}
45       || error(string(C) + " isn't caught")
46     }
47   }
48 }

```

Figure 4.10: Checked exceptions

4.5 Race Detection

Type-based race detection as proposed by Flanagan [FF00] is another example of a type-and-effect system. The “effect” is a lock. Each field is annotated with the lock that guards it, and each method is annotated with the set of locks it requires. To avoid alias analyses, this type system requires that all locks be given by final expressions so that lock equivalence is the same as structural equality of ASTs. Flanagan also describes classes parameterized by locks and thread-local classes; these features are not yet implemented in our extension.

The important syntax, scoperules, and typerules for type-based race detection are shown in figure 4.11. Again, this extension is very straightforward. Like the previous extension, lock sets are stored as type annotations. Not shown are the “final” module which implements final annotations and the typerules which ensure that all declared guards are final expressions. The function `checklocksubset` checks if one set of locks is a subset of another, and if not, prints an error message. On line 54, the function `locksetsubst` is used to substitute `this` into the lockset which guards a class field or method.

```

1  require "std";
2  require "annotate";
3  require "final";
4
5  extend grammar Java
6  {
7    field: usertype name "guarded_by" expr ";" ;
8    proto: proto "requires" expr[ , ]+ ;
9
10   stmt: "synchronized" expr "in" stmt
11        | "fork" stmt ;
12
13   annotation: "guarded_by" expr[ , ]+ ;
14 }
15
16 scoperules
17 {
18   lockset: list Java.expr ;
19
20   ~proto{ \proto requires \ls } {
21     term.lockset@out = listconcat(ls, default) ;
22   }
23   ~method{ \proto \body } {
24     term.lockset@out = term.lockset@in ;
25   }
26
27   ~stmt{ synchronized \l in \s } {
28     s.lockset@in      = l # default ;
29     term.lockset@out = term.lockset@in ;
30   }
31
32   ~stmt{ fork \s } {
33     s.lockset@in      = [] ;
34     term.lockset@out = term.lockset@in ;
35   }
36 }
37
38 typerules
39 {
40   restrict
41   ~expr{ this } :: _ {
42     exists(l in term.lockset)
43       {astequal(l, term)}
44     || error(string(C) + " isn't caught")
45   }
46
47   restrict
48   ~expr{ \x::name } :: { @(guarded_by \ls) \T } {
49     checklocksubset(term.line, ls, term.lockset)
50   }
51
52   restrict
53   ~expr{ \e . \m } :: { @(guarded_by \ls) \T } {
54     checklocksubset(term.line, ls, locksetsubst(term.lockset, e, '{this}'))
55   }
56 }

```

Figure 4.11: Type-based race detection

CHAPTER 5

Conclusion

Programmers have long wished to extend or modify their languages' type systems, but have had few easy ways to do so. Typmix is a framework that tries to fill this void. The typerules language provides a concise and familiar notation for declaring typing judgements. The scoperules language provides a concise notation for describing how scopes propagate. The separation of these two languages proved to be an important and useful design decision.

This work aims to make it easy to extend and modify type systems. Case studies on two languages suggest that Typmix is an easy-to-use framework. These languages were selected because they cover a diverse range of application, from inference-heavy functional languages to imperative object-oriented languages. Further, the extensions studied represent recent academic research as well as examples from real-world languages. From these case studies, we conclude that Typmix remains relatively easy to use, even when the type system being implemented is non-trivial.

We hope that extensible language frameworks like Typmix will one day become a standard part of the development tool chain.

REFERENCES

- [ANM06] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. “A framework for implementing pluggable type systems.” In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pp. 57–74, New York, NY, USA, 2006. ACM Press.
- [Bac98] John Backus. “The History of Fortran I, II, and III.” *IEEE Annals of the History of Computing*, **20**(4):68–78, 1998.
- [BOS98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. “Making the future safe for the past: adding genericity to the Java programming language.” In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 183–200, New York, NY, USA, 1998. ACM Press.
- [Bra04] Gilad Bracha. “Pluggable type systems.” In *OOPSLA Workshop on the Revival of Dynamic Languages*, 2004.
- [CBCon] Russ Cox, Tom Bergan, Austin Clemens, Frans Kaashoek, and Eddie Kohler. “Xoc: And Extension-Oriented Compiler for Systems Programming.” In *ASPLOS 2008*, under submission.
- [CHA07] Jeremy Condit, Matthew Harren, Zachary R. Anderson, David Gay, and George C. Necula. “Dependent Types for Low-Level Programming.” In Rocco De Nicola, editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pp. 520–535. Springer, 2007.
- [CMM05] Brian Chin, Shane Markstrum, and Todd Millstein. “Semantic type qualifiers.” In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pp. 85–95, New York, NY, USA, 2005. ACM Press.
- [Coq06] The Coq Development Team. “The Coq Proof Assistant Reference Manual.” <http://coq.inria.fr/V8.1/refman/index.html>, 2006.
- [Cre97] Roger F. Crew. “ASTLOG: a language for examining abstract syntax trees.” In *DSL'97: Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL)*, 1997, pp. 18–18, Berkeley, CA, USA, 1997. USENIX Association.
- [DF01] Robert DeLine and Manuel Fähndrich. “Enforcing high-level protocols in low-level software.” In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pp. 59–69, New York, NY, USA, 2001. ACM Press.
- [DM82] Luis Damas and Robin Milner. “Principal type-schemes for functional programs.” In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 207–212, New York, NY, USA, 1982. ACM Press.

- [FF00] Cormac Flanagan and Stephen N. Freund. “Type-based race detection for Java.” In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pp. 219–232, New York, NY, USA, 2000. ACM Press.
- [FFA99] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. “A theory of type qualifiers.” In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pp. 192–203, New York, NY, USA, 1999. ACM Press.
- [GJS05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, 2005.
- [GJS06] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. “Concepts: linguistic support for generic programming in C++.” In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pp. 291–310, New York, NY, USA, 2006. ACM Press.
- [God97] Patrice Godefroid. “Model checking for programming languages using VeriSoft.” In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 174–186, New York, NY, USA, 1997. ACM Press.
- [Hej03] Bill Venners and Bruce Eckel. “A Conversation with Anders Hejlsberg, Part II.” <http://www.artima.com/intv/handcuffs.html>, August 2003.
- [How80] William A. Howard. “The formulas-as-types notion of construction.” In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pp. 479–490. Academic Press, 1980. Reprint of 1969 article.
- [IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. “Featherweight Java: a minimal core calculus for Java and GJ.” *ACM Trans. Program. Lang. Syst.*, **23**(3):396–450, 2001.
- [Joh87] Thomas Johnsson. “Attribute grammars as a functional programming paradigm.” In *Proc. of a conference on Functional programming languages and computer architecture*, pp. 154–173, London, UK, 1987. Springer-Verlag.
- [Jon93] Mark P. Jones. “A system of constructor classes: overloading and implicit higher-order polymorphism.” In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*, pp. 52–61, New York, NY, USA, 1993. ACM Press.
- [Jon03] Simon Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge Press, 2003.
- [Knu68] Donald E. Knuth. “Semantics of Context-Free Languages.” *Mathematical Systems Theory*, **2**(2):127–145, 1968.
- [LCH07] Daniel K. Lee, Karl Crary, and Robert Harper. “Towards a mechanized metatheory of standard ML.” In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT*

symposium on Principles of programming languages, pp. 173–184, New York, NY, USA, 2007. ACM Press.

- [Ler06] Xavier Leroy. “Formal certification of a compiler back-end or: programming a compiler with a proof assistant.” In *POPL ’06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 42–54, New York, NY, USA, 2006. ACM Press.
- [LP03] Michael Y. Levin and Benjamin C. Pierce. “TinkerType: A Language for Playing with Formal Systems.” *Journal of Functional Programming*, **13**(2), March 2003.
- [Mil78] Robin Milner. “A Theory of Type Polymorphism in Programming.” *Journal of Computer and System Sciences*, **17**(3):348–375, 1978.
- [ML00] Andrew C. Myers and Barbara Liskov. “Protecting privacy using the decentralized label model.” *ACM Trans. Softw. Eng. Methodol.*, **9**(4):410–442, 2000.
- [NCH05] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. “CCured: type-safe retrofitting of legacy software.” *ACM Trans. Program. Lang. Syst.*, **27**(3):477–526, 2005.
- [NCM03] Nathaniel Nystrom, Michael Clarkson, and Andrew Myers. “Polyglot: an Extensible Compiler Framework for Java.” In *Proceedings of the 12th International Conference on Compiler Construction*, April 2003.
- [NMR02] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. “CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs.” In *CC ’02: Proceedings of the 11th International Conference on Compiler Construction*, pp. 213–228, London, UK, 2002. Springer-Verlag.
- [NQM06] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. “J&: nested intersection for scalable software composition.” In *OOPSLA ’06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pp. 21–36, New York, NY, USA, 2006. ACM Press.
- [Ode93] Martin Odersky. “Defining context-dependent syntax without using contexts.” *ACM Trans. Program. Lang. Syst.*, **15**(3):535–562, 1993.
- [Paa95] Jukka Paakki. “Attribute grammar paradigm: a high-level methodology in language implementation.” *ACM Computing Surveys*, **27**(2):196–255, 1995.
- [Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
- [PS99] Frank Pfenning and Carsten Schürmann. “System Description: Twelf - A Meta-Logical Framework for Deductive Systems.” In *CADE-16: Proceedings of the 16th International Conference on Automated Deduction*, pp. 202–206, London, UK, 1999. Springer-Verlag.

- [Rem92] Didier Rémy. “Extending ML Type System with a Sorted Equational Theory.” Research Report 1766, Institut National de Recherche en Informatique et Automatisation, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, 1992.
- [ST06] Jeremy G. Siek and Walid Taha. “Gradual typing for functional languages.” In *Scheme and Functional Programming Workshop*, September 2006.
- [Vis97] Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.
- [Vis04] Eelco Visser. “Program Transformation with Stratego/XT. Rules, Strategies, Tools, and Systems in Stratego/XT 0.9.” Technical Report UU-CS-2004-011, Institute of Information and Computing Sciences, Utrecht University, 2004.
- [VKS07] Eric Van Wyk, Lijesh Krishnan, August Schwerdfeger, and Derek Bodin. “Attribute Grammar-based Language Extensions for Java.” In *European Conference on Object Oriented Programming (ECOOP)*, Lecture Notes in Computer Science. Springer Verlag, July 2007.
- [WB89] P. Wadler and S. Blott. “How to make ad-hoc polymorphism less ad hoc.” In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 60–76, New York, NY, USA, 1989. ACM Press.
- [Wri95] Andrew K. Wright. “Simple imperative polymorphism.” *Lisp Symb. Comput.*, **8**(4):343–355, 1995.
- [XP98] Hongwei Xi and Frank Pfenning. “Eliminating array bound checking through dependent types.” In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pp. 249–257, New York, NY, USA, 1998. ACM Press.