Constraint-Based Polymorphism in Cecil

Vassily Litvinov and Craig Chambers

Department of Computer Science and Engineering
University of Washington, Box 352350
Seattle, Washington 98195-2350 USA

Technical Report UW-CSE-98-01-01
January 1998

{vass,chambers}@cs.washington.edu
(206) 685-2094; fax: (206) 543-2969

Constraint-Based Polymorphism in Cecil

Vassily Litvinov and Craig Chambers

Department of Computer Science and Engineering
University of Washington

{vass,chambers}@cs.washington.edu

UW CSE Technical Report UW-CSE-98-01-01

Abstract

We are developing a static type system for object-oriented languages that strives to guarantee statically and completely the absence of certain large classes of run-time errors, to enable polymorphic abstractions to be written and typechecked once separately from any current or future instantiations, and to avoid forcing code to be written unnaturally due to static type system limitations. The type system supports bounded parametric polymorphism, where the bounds on type variables can be expressed using a mixture of recursive subtype and signature constraints; this kind of bounding supports F-bounded polymorphism, Theta-style `where` clauses, and covariant and contravariant type parameters as special cases. The type system coexists with many advanced language features, including multi-methods, independent inheritance and subtyping, mutable and immutable state, first-class lexically-scoped functions, and mixed statically and dynamically typed code. We have implemented this type system in the Cecil language, and we have used it to successfully typecheck a 100,000-line Cecil program, the Vortex optimizing compiler.

1 Introduction

Static typechecking offers several advantages to programmers, including early and possibly complete detection of some kinds of programming errors, support for other static reasoning about programs by both people and machines, and potentially easier or more effective implementation. However, if the type system is too simple or restrictive, static typechecking can require unnecessary duplication of code, force code to be written in unnatural styles, disallow some useful programming constructions, force explicit type “casts” to be inserted to work around limitations in the static typechecker (with attendant run-time cost if checked), and/or lead to static type systems with incomplete checking or other type loopholes.

We are developing a static type system for object-oriented languages whose goal is guarantee statically the absence of certain classes of run-time errors while reducing as much as possible the problems of strict static typechecking listed above. Our type system has the following salient characteristics:

- The type system supports **bounded parametric polymorphism** in addition to the normal subtype polymorphism of object-oriented languages. Polymorphic abstractions are written and typechecked once, not repeatedly for each instantiation or for each subclass.
- The bounds on the allowed polymorphism of type variables can be expressed using a mixture of recursive **subtype** and **signature constraints**, supporting both F-bounded polymorphism [Canning et al. 89] and Theta-style `where` clauses [Day et al. 95, Liskov et al. 94] as special cases. Constraints over subtyping declarations enable expression of conditional subtyping as well as co- or contravariant subtyping between different instances of the same parameterized type.

- The type system supports many advanced language features, including multi-methods, independent inheritance and subtyping, mutable and immutable state, first-class lexically-scoped functions, classless object models, and mixed statically and dynamically typed code.

We have implemented this type system in the Cecil language [Chambers 92, Chambers 93a], and we have used its features to successfully typecheck a 100,000-line Cecil program, the Vortex optimizing compiler system [Dean et al. 96] (which includes the Cecil typechecker itself as a component).

The next section of this paper presents our host language and the type system informally, illustrating the capabilities of our type system on several examples, and comparing it to other type systems. Section 3 specifies the type system more formally. Section 4 reports on our experience in using the type system in typechecking Vortex. Section 5 compares our work to other type systems.

2 Informal Description of the Type System

In this section we informally describe our polymorphic type system. We begin with a brief description of the host language that forms the context for our work, then proceed to describe aspects of the polymorphic type system, in increasing levels of sophistication. Section 3 specifies our type system more formally.

We define a simplified version of Cecil, called Mini-Cecil, for use in presenting our type system and giving examples. Mini-Cecil focuses our attention on the essential constructs relevant to the expressiveness of our type system, omitting features of Cecil that are orthogonal to this focus.* Figure 1 specifies the abstract syntax of Mini-Cecil. The polymorphic component of the language is confined to the *Context* part of declarations and to the type parameters [$\overline{T}$] associated with all named entities other than variables. The following subsection provides background by describing the monomorphic core of Mini-Cecil ignoring these polymorphic features; a version of this monomorphic language and the issues involved in typechecking it have been discussed previously [Chambers & Leavens 94, Chambers & Leavens 95]. Later subsections reintroduce the polymorphic features.

2.1 Monomorphic Host Language

A Mini-Cecil program is a collection of (mutually recursive, unordered) declarations followed by an expression to evaluate. Classes and types are independent notions in Mini-Cecil, and different aspects of a class or type are defined through separate declarations. The **class** *c* declaration introduces a new class named *c* (which may either be **abstract** and potentially incomplete, or **concrete** and instantiable by **new** expressions), the c_1 **inherits** c_2 declaration states that c_1 is a subclass of c_2 (multiple inheritance is possible), and the **method** and **field** declarations add methods and mutable instance variables, respectively, to classes. Similarly, the **type** *t* declaration introduces a new type named *t*, the T_1 **subtypes** T_2 declaration states that T_1 is a subtype of T_2 (multiple subtyping is possible), and the **sig** $m(T_1,...,T_n):T_r$ declaration adds an operation named *m* to the interfaces of the $T_1,...,T_n$ argument types, returning T_r. The *c* **conforms** *T* declaration connects the class hierarchy to the type hierarchy, stating that direct instances of class *c* (but not necessarily instances of its subclasses) support the operations of type *T* (and all *T*'s supertypes). The **var** *v*:*T* declaration declares a new mutable variable named *v* holding values of type *T*.

Mini-Cecil is based on multiple dispatching; multi-methods generalize undispatched procedures, singly-dispatched methods, and functions overloaded on different argument types, and multi-methods can resolve

* Omitted features include a classless object model, predicate objects [Chambers 93b], initialized variable and field declarations, initialization of fields as part of object creation, immutable variables and fields, invocation of function objects via messages, nested declarations other than variables, modules and encapsulation, analogues of Smalltalk's `super` and non-local return constructs [Goldberg & Robson 83], mixed dynamically and statically typed code, and various syntactic conveniences such as infix and dot notation and simultaneous declaration of parallel inheritance and subtyping hierarchies.

Variable	v		
Class	c		
Type Name	t		
Type Variable	α		
Message	m		
Program	P	::=	$\overline{CD}\ \overline{VD}\ E$
Constrained Decl	CD	::=	$CX\ D$
Context	CX	::=	**forall** $\overline{\alpha}$ **where** $\overline{C}$:
Constraint	C	::=	$SubD \mid SigD$
Declaration	D	::=	$ClassD \mid InhD \mid MethD \mid FieldD \mid TypeD \mid SubD \mid SigD \mid ConfD$
Class Decl	$ClassD$	::=	$role$ **class** $c[\overline{T}]$
Inherits Decl	$InhD$	::=	$c_1[\overline{T}_1]$ **inherits** $c_2[\overline{T}_2]$
Method Decl	$MethD$	::=	**method** $m[\overline{T}_p](\overline{v@c:T}):T_r\ \{\ B\ \}$
Field Decl	$FieldD$	::=	**field** $m[](v@c:T):T_r$
Type Decl	$TypeD$	::=	**type** $t[\overline{T}]$
Subtype Decl	$SubD$	::=	T_1 **subtypes** T_2
Signature Decl	$SigD$	::=	**sig** $m\ [\overline{T}_p](\overline{T}_a):T_r$
Conforms Decl	$ConfD$	::=	$c[\overline{T}]$ **conforms** T_2
Class Role	$role$	::=	**abstract** | **concrete**
Type	T	::=	$t[\overline{T}] \mid \lambda(\overline{T}):T_r \mid \alpha \mid$ **glb**$(\overline{T}) \mid$ **lub**$(\overline{T})$
Block	B	::=	$\overline{VD}\ E$
Var Decl	VD	::=	**var** $v:T$
Expression	E	::=	$v \mid v := E \mid$ **new** $c[\overline{T}] \mid m\ [\overline{T}_p](\overline{E}_a) \mid \lambda(\overline{v:T}):T_r\ \{\ B\ \} \mid$ **apply**$(E_\lambda\ \overline{E}) \mid E_1\ ;\ E_2$

We use overbar notation, as in $\overline{T}$, to indicate a sequence of zero or more elements. In program examples, we separate these elements by commas or semicolons. We also omit empty constructs where desired, including empty type parameter lists "[]", contexts with no type variables, and empty sets of constraints.

Figure 1: Syntax of Mini-Cecil.

classic challenges such as the "binary methods" problem [Bruce et al. 95a]. Multi-methods are associated with classes by having their formal parameters specialized for particular classes; a formal parameter declaration $v@c:T$ introduces a formal parameter named v of type T that is specialized on the class c. (The effect of an unspecialized formal may be achieved by specializing on the `Any` class from which all other classes inherit.) A method applies to a message if all the actual parameters to the send are instances of classes that descend from the corresponding specializer classes. One method overrides another if its tuple of specializer classes descends from the other's tuple, comparing tuples pointwise (CLOS compares tuples lexicographically; the choice is uninteresting for our type system work). Static typechecking must ensure that every message at run-time will locate a single most-specific applicable method.

Field declarations introduce new mutable instance variables, accessed solely through automatically-generated get and set accessor methods. A field declaration **field** $m(v@c:T):T_r$ declares a field named m for all instances of the specializer class c or any of its subclasses. The field declaration automatically declares a get accessor method **method** $m(v@c:T):T_r$ that returns the contents of the m field of its argument object, and a set accessor method **method** $m(v@c:T, contents@\texttt{Any}:T_r):\texttt{Void}$ that takes an object and a new value and updates the m field of the object to hold the new value.

Mini-Cecil has standard expressions for variable reference, assignment, and statement sequencing. The **new** c expression creates an instance of the class c. The $m(E_1,...,E_n)$ expression sends the m message to its n argument objects. The $\lambda(v_1:T_1,...,v_n:T_n):T_r\{B\}$ expression creates a lexically-nested first-class function

object, taking *n* arguments named $v_1,...,v_n$ of type $T_1,...,T_n$, whose body is the block *B* returning a value of type T_r. The **apply**($E_\lambda,E_1,...,E_n$) expression invokes its first argument (a function object created by a λ expression), passing the remaining arguments. The $\lambda(T_1,...,T_n):T_r$ form names the type of a function object taking arguments of types $T_1,...,T_n$ and returning a value of type T_r. The standard contravariant subtyping rules hold for function types: one function type $\lambda(T_1,...,T_n):T_r$ is implicitly a subtype of another function type $\lambda(T_1',...,T_n'):T_r'$ if each T_i is a supertype of T_i' and T_r is a subtype of T_r'. (First-class functions are used in Mini-Cecil to express control structures within the language.)

The following example illustrates these declarations and expressions using a simple `Point`/`ColorPoint` hierarchy. (Note that the "binary method" problem with the `equal_point` method on colored points is resolved through multiple dispatching.)

```
-- Interfaces:
type Point;
  sig x(Point):Num;
  sig set_x(Point, Num):Void;
  sig y(Point):Num;
  sig set_y(Point, Num):Void;
  sig area(Point):Num;
  sig equal_point(Point, Point):Bool;
  sig new_point(Num, Num):Point;
type ColorPoint;
  ColorPoint subtypes Point;
  sig color(ColorPoint):Color;
  sig set_color(ColorPoint, Color):Void;
  sig new_color_point(Num, Num, Color):ColorPoint;
-- Implementations:
concrete class PointClass;
  PointClass inherits Any;
  PointClass conforms Point;
  field x(p@PointClass:Point):Num;  -- introduces x and set_x methods
  field y(p@PointClass:Point):Num;  -- introduces y and set_y methods
  method area(p@PointClass:Point):Num { multiply(x(p), y(p)) }
  method equal_point(p1@PointClass:Point, p2@PointClass:Point):Bool {
    and(equal_num(x(p1), x(p2)), λ(){ equal_num(y(p1), y(p2)) }) }
  method new_point(x0@Any:Num, y0@Any:Num):Point {
    var result:Point;
    result := new PointClass;
    set_x(result, x0);
    set_y(result, y0);
    result }
concrete class ColorPointClass;
  ColorPointClass inherits PointClass;
  ColorPointClass conforms ColorPoint;
  field color(p@ColorPointClass:ColorPoint):Color;  -- introduces color and set_color methods
  method equal_point(p1@ColorPointClass:ColorPoint,
                     p2@ColorPointClass:ColorPoint):Bool {
    and(and(equal_num(x(p1), x(p2)), λ(){ equal_num(y(p1), y(p2)) }),
        λ(){ equal_color(color(p1), color(p2)) }) }
  method new_color_point(x0@Any:Num, y0@Any:Num, c0@Any:Color):ColorPoint {
    var result:ColorPoint;
    result := new ColorPointClass;
    set_x(result, x0);
    set_y(result, y0);
    set_color(result, c0);
    result }
```

Note that this syntax is verbose in several ways: declaring separate but parallel inheritance and subtyping, using prefix message notation to read and write fields and for "infix" operators, explicit first-class functions for control structures (such as the short-circuiting `and` method used above), and requiring `Any` as the specializer class for many formals. The full Cecil language has syntactic sugars that make these common patterns concise, without changing the underlying semantics. In this paper, we will stick to the unsugared syntax of Mini-Cecil.

Given the type partial order derived from programmer-specified **type** and **subtypes** declarations, the system automatically completes the lattice to provide least-upper- and greatest-lower-bounds for all pairs of types. The **lub**(T_1,...,T_n) and **glb**(T_1,...,T_n) forms name the least-upper-bound and greatest-lower-bound, respectively, of a list of types. Unless one type is a subtype of the other, the least-upper-bound and the greatest-lower-bound of two types are different from any common ancestor or descendant of the types. For example, **lub**(`Int`,`Float`) is a strict subtype of any supertype that `Int` and `Float` have in common, like `Num`.

Typechecking of this monomorphic core of Mini-Cecil divides into two components: client-side typechecking and implementation-side typechecking. Client-side typechecking uses the interfaces declared through **type**, **subtypes**, **conforms**, and **signature** declarations to verify that expressions are type-correct. These checks include that the type of the right-hand-side of an assignment is a subtype of the type of the left-hand-side, and that the type of the body of a method or function object is a subtype of the declared return type. The type of a **new** *c* expression is the **glb** of all the types to which *c* conforms. A message expression $m(E_1,...,E_n)$, whose arguments have types $T_1,...,T_n$, is type-correct iff there exists a covering **sig** $m(T_1',...,T_n'):T_r'$ declaration where each of the T_i is a subtype of the corresponding T_i'. The type of the result of such a message is the **glb** of the T_r' of all covering signatures. The type of a $\lambda(v_1:T_1,...,v_n:T_n):T_r\{B\}$ expression is $\lambda(T_1,...,T_n):T_r$. An **apply**(E_λ,E_1,...,E_n) expression, whose arguments have types T_λ,T_1,...,T_n, is type-correct iff there exists a covering function type $\lambda(T_1',...,T_n'):T_r'$ that is a supertype of T_λ and where each T_i is a subtype of the corresponding T_i'. The type of the result of such an application is the **glb** of the T_r' of all covering function types.

Implementation-side typechecking verifies that the implementations given by **class**, **inherits**, **method**, and **field** declarations support the interface declared through **type**, **subtypes**, and **signature** declarations, as linked through the **conforms** declarations. In effect, these checks enumerate, for each signature **sig** $m(T_1,...,T_n):T_r$, all *n*-tuples of **concrete** argument classes $c_1,...,c_n$ that conform to the corresponding signature types. For each of these tuples, typechecking verifies that a single most-specific applicable method **method** $m(v_1'@c_1':T_1',...,v_n'@c_n':T_n'):T_r'$ (which may be a field accessor method) would be looked up, and that moreover each c_i argument class conforms to the corresponding declared formal type T_i', and that the method's result type T_r' is a subtype of the signature's result type T_r. (An efficient algorithm to achieve the effect of this enumeration-based specification has been described previously [Chambers & Leavens 94, Chambers & Leavens 95]). If these tests succeed, then the type interface is known to be completely and unambiguously implemented. If client-side typechecking also succeeds, overall type-safety of the program is assured.

2.2 Polymorphic Declarations

To support parametric polymorphism, any global declaration (other than variables) may be polymorphic over some set of types by prefixing the declaration with a **forall** $\alpha_1,...,\alpha_n$ context. The α_i are type variables over which the declaration is polymorphic, and they may be used as regular types within the prefixed declaration. (A non-polymorphic declaration has an empty list of type variables.)

To use a polymorphic declaration, it must first be instantiated by providing actual types for each type variable. Instantiating types for type variables may be provided explicitly by clients of the declaration or inferred implicitly by the system. A polymorphic declaration specifies the set of explicitly-passed type parameters by providing a list of type variables in brackets after the name of the declared entity. For instance, in the polymorphic class declaration **forall** $\alpha_1,...,\alpha_n$: **class** $c[\alpha_i,...,\alpha_j]$, the $\alpha_i,...,\alpha_j$ type parameters are provided explicitly by clients. Uses of the declared name provide a list of actual types to use for those type variables, for instance **new** $c[T_i,...,T_j]$.

The following declarations use these mechanisms to define a parameterized read-only array class and type. (The full Cecil language includes syntactic sugar to make the **forall** clauses implicit in most circumstances.) In our examples we use `T` and `S` to name type variables.

```
forall T: type Array[T]
forall T: Array[T] subtypes Collection[T]
forall T: sig fetch(Array[T], Int):T

forall T: concrete class ArrayClass[T]
forall T: ArrayClass[T] inherits CollectionClass[T]
forall T: ArrayClass[T] conforms Array[T]
forall T: method fetch(a@ArrayClass[T]:Array[T], index@Any:Int):T  { ... }

var my_array:Array[Num] := new ArrayClass[Num];
var result:Num := fetch(my_array, 5);
```

The `Array` type and the `ArrayClass` class are both explicitly parameterized. All references to these names must provide instantiating types, such as type annotations like `Array[Num]` and instance creation expressions like **new** `ArrayClass[Num]`. The instantiations `Array[T]` and `ArrayClass[T]` in the other declarations use the type variable `T` as the instantiating type; `T` is bound by the **forall** prefix and is considered a regular type (of unknown properties) in the scope of the **forall**.

In contrast, the **sig** and **method** declarations for `fetch` are implicitly parameterized. References to `fetch` do not pass any explicit type parameters. Instead, the system infers the appropriate type parameter from the types of the actual parameters. For example, in the `fetch(my_array, 5)` expression, because `my_array` is of type `Array[Num]`, the type parameter `T` in the `fetch` signature is inferred automatically to be `Num`, since that is the most precise instantiating type that makes the expression typecheck. Automatic inference of instantiating type parameters is a key feature of our type system and an important contributor to its convenience in practice.

The **subtypes**, **inherits**, and **conforms** declarations reference previously declared parameterized types and/or classes, and state that a particular relationship holds between particular instances of the types and/or classes for a certain pattern of instantiating types. Clients do not directly reference these anonymous declarations; instead the system automatically instantiates them as needed to support its run-time method lookup and typechecking operations.

Our type system's polymorphism model supports dependent types, where the type of one argument or result depends on or is computed from aspects of the type(s) of other arguments. The `fetch` method above is an example of this, where the type of the result of `fetch` is dependent on the array element type. Other examples occur with control structures, with the types of the argument function objects inducing other types. As a simple example, the following declarations illustrate how the polymorphic `if` expression is programmed in Mini-Cecil, following the model of `ifTrue:ifFalse:` in Smalltalk [Goldberg & Robson 83].

```
type Bool;
  forall T: sig if(Bool, λ():T, λ():T):T
concrete class True;
  True conforms Bool;
  forall T: method if(t@True:Bool, if_true@Any:λ():T, if_false@Any:λ():T):T {
    apply(if_true) }
concrete class False;
  False conforms Bool;
  forall T: method if(f@False:Bool, if_true@Any:λ():T, if_false@Any:λ():T):T {
    apply(if_false) }
```

In this example, the result of the `if` message is inferred from the result types of the two function arguments. If the two function objects have different result types, the system automatically will search for the single most-specific type to bind to `T` that will make the call type-correct; such a type will always exist because the least-upper-bound of all pairs of unrelated types has been introduced automatically. For example, the type bound to `T` (and the result of the message) in the following expression is inferred to be `Int|Float`.

```
... if(..., λ(){ 3 }, λ(){ 4.5 }) ...
```

2.3 Bounded Polymorphism

It is often necessary to restrict the polymorphism of a declaration to types that have some property, such as that values of the type be printable or comparable. This bounded polymorphism is supported in our type system by placing one or more constraints on the type variables, in the **where** clause of the **forall** prefix. A constraint may either be that one type expression can be shown to be a subtype of another, or that a particular signature can be known to hold. Constraints on the type parameters of a **method** declaration are exploited directly during typechecking of the body of method. Constraints on the type parameters of **class** and **type** declarations restrict allowed instantiating types, which indirectly enables methods and signatures associated with the parameterized classes and types to exploit these constraints. Constraints on **inherits**, **subtypes**, and **conforms** declarations restrict when these relationships are known, and must be verified, to hold.

The following declarations use subtype-bounded polymorphism to define a method to print out collections, for those collections whose elements support the `print` operation. The `print` operation on collections is typechecked once, ensuring that it will be type-safe for all instantiating types that are subtypes of `Printable`.

```
type Printable;
  sig print(Printable):Void;
String subtypes Printable;  -- and other previously declared types, too
forall T where T subtypes Printable:
  Collection[T] subtypes Printable;  -- collections of printable things are themselves printable
forall T where T subtypes Printable:
  method print(a@CollectionClass[T]:Collection[T]):Void {
    print("[");
    do(a, λ(e:T){ print(e) });  -- do invokes its function argument on each element of the collection a
    print("]") }
```

Alternatively, signature-bounded polymorphism may be used to declare such a `print` operation on collections, without introducing an explicit `Printable` type, as illustrated by the following declaration.

```
forall T where sig print(T):Void:
  method print(a@CollectionClass[T]:Collection[T]):Void {
    print("[");
    do(a, λ(e:T){ print(e) });  -- do invokes its function argument on each element of the collection a
    print("]") }
```

The signature-based style is more compact in this example and it avoids the need for declaring explicitly which types are subtypes of `Printable`. In effect, a signature constraint supports a kind of automatically inferred structural subtype constraint. The subtype-based style builds on an explicit named subtyping

model. Subtype constraints from explicitly named types can work better where several signatures are associated with one type, where the concept embodied by the bound merits an explicit name, or where accidental structural matches should be avoided. Our polymorphism mechanism does not force programmers to choose one approach over the other; both kinds of bounds are supported equally.

2.4 Recursive Constraints

A type variable may appear in multiple constraints and in different parts of the same constraint. In addition, because the polymorphic declarations being introduced are visible throughout the enclosing scope, they may be referenced in the constraints themselves. These features enable our system to express F-bounded polymorphism and Theta-style `where` clauses easily. (Additionally, our system can express mixes of the two styles.) The following declarations use this idiom to declare the types and default implementations of things that are comparable and totally-ordered. (To avoid repeating the same **forall** clauses for a series of declarations, we introduce a syntactic sugar of the form **forall** α **where** *C*`:` `{` *D*$_1$; ...; *D*$_n$ `}`, which is the same as **forall** α **where** *C*`:` *D*$_1$; ...; **forall** α **where** *C*`:` *D*$_n$. If any of the *D*$_i$ has a **forall** or **where** clause of its own, its clause is merged with the enclosing **forall** as part of desugaring. Full Cecil includes other syntactic sugars that minimize the verbosity of **forall** clauses.)

```
forall T where T subtypes Comparable[T]: {
  type Comparable[T]
    sig equal(T, T):Bool
    sig not_equal(T, T):Bool
  abstract class ComparableClass[T]
    ComparableClass[T] conforms Comparable[T]
    method not_equal(x1@ComparableClass[T]:T, x2@ComparableClass[T]:T):Bool {
      not(equal(x1, x2)) }
}
forall T where T subtypes Ordered[T]: {
  type Ordered[T]
    Ordered[T] subtypes Comparable[T]
    -- interfaces for equal and not_equal are "inherited" by Ordered
    sig less_than(T, T):Bool
    sig less_or_equal(T, T):Bool
    sig greater_or_equal(T, T):Bool
    sig greater_than(T, T):Bool
    sig min(T, T):T
    sig max(T, T):T
    forall S: sig compare_and_do(T, T, λ():S, λ():S, λ():S):S
  abstract class OrderedClass[T]
    OrderedClass[T] inherits ComparableClass[T]
    OrderedClass[T] conforms Ordered[T]
    method less_or_equal(x1@OrderedClass[T]:T, x2@OrderedClass[T]:T):Bool {
      or(equal(x1, x2), λ(){ less_than(x1, x2) }) }
    -- default methods for greater_or_equal and greater_than are similar
    method min(x1@OrderedClass[T]:T, x2@OrderedClass[T]:T):T {
      if(less_or_equal(x1, x2), λ(){ x1 }, λ(){ x2 }) }
    -- default method for max is similar
    forall S:
      method compare_and_do(x1@OrderedClass[T]:T, x2@OrderedClass[T]:T,
                            if_less:λ():S, if_equal:λ():S, if_greater:λ():S):S {
        if(less_than(x1, x2), if_less, λ(){ if(equal(x1, x2), if_equal, if_greater) }) }
}
```

In this example, the `Comparable` and `Ordered` types define a number of operations available on implementations conforming to the type. None of these operations requires explicit type parameters, and so clients can simply invoke the operations directly. The `ComparableClass` and `OrderedClass` classes provide default implementations of most of these operations, from which classes that wish to be comparable or ordered may inherit. (Using signature constraints to require several operations on client ordered values

would be more verbose than using a single subtype constraint from the `Ordered` type.) The `min`, `max`, and `compare_and_do` methods have their type parameters and result type inferred by the system from the types of their arguments.

By bounding the type parameter `T` in terms of itself (an example of F-bounded polymorphism), this code ensures that the two values being tested come from the same "domain," providing one solution to the "binary methods" problem (multi-methods are an alternative solution which resolves the problem with a different final semantics). For example, in the following declarations, the bound will ensure that numbers can be compared to numbers, and collections of numbers can be compared to other collections of numbers, but will disallow comparing numbers to collections of numbers, unlike covariant typing as in Eiffel [Meyer 92] or virtual types as in Beta [Madsen & Møller-Pederse 89, Madsen et al. 93] and Thorup's proposal for Java [Thorup 97]. Additionally, different representations of numbers (such as integers and floats) can be intermixed in the collection and compared to each other; subtyping is still available after instantiating the polymorphic declarations, unlike LOOM [Bruce et al. 97] and Haskell [Wadler & Blott 89].

```
type Num
  Num subtypes Ordered[Num]  -- classes conforming to Num can be compared to each other
abstract class NumClass
  NumClass conforms Num
  NumClass inherits OrderedClass[Num]  -- inherit default implementations of operations
  sig convert_to_float(Num):Num
  method equal(x1@NumClass:Num, x2@NumClass:Num):Bool {
    equal(convert_to_float(x1), convert_to_float(x2)) }
  -- less_than method is similar
concrete class IntClass
  IntClass inherits NumClass
  IntClass conforms Num
  method equal(x1@IntClass:Num, x2@IntClass:Num):Bool { ... }
  -- less_than method is similar
concrete class FloatClass
  FloatClass inherits NumClass
  FloatClass conforms Num
  method equal(x1@FloatClass:Num, x2@FloatClass:Num):Bool { ... }
  -- less_than method is similar
forall T: {
  type Collection[T]
  where T subtypes Ordered[T]:
    Collection[T] subtypes Ordered[Collection[T]]  -- lexicographic ordering
  sig length(Collection[T]):Int
  sig do(Collection[T], λ(T):Void):Void
  forall S: sig pairwise_do(Collection[T], Collection[S], λ(T,S):Void):Void
}
forall T: {
  abstract class CollectionClass[T]
    CollectionClass[T] conforms Collection[T]
    where T subtypes Ordered[T]: {
      CollectionClass[T] inherits OrderedClass[Collection[T]]
      method equal(c1@CollectionClass[T]:Collection[T],
                   c2@CollectionClass[T]:Collection[T]):Bool {
        and(equal(length(c1), length(c2)),
            λ(){ pairwise_do(c1, c2, λ(e1:T, e2:T){
                   if(not_equal(e1, e2), λ(){ return False }, λ(){ VoidValue })
                 });
                 True }) }
      -- less_than method is similar
    }
}
```

2.5 Constraint Solving

The heart of our polymorphic type system is a constraint solver. All type checks are expressed in terms of a system of subtype, conforms, and signature constraints over types which may contain unbound type variables, along with conditions for each unbound type variable as to whether the most specific, most general, or any legal instantiating type is to be computed. The system of constraints is handed off to a constraint solver to be solved, in the context of *assumed* subtype, conforms, and signature constraints. Assumed constraints include the global subtype, conforms, and signature declarations and the subtype and signature constraints from the enclosing **where** clause, if any. The solver attempts to compute types for the unbound type variables (if any) and show that the system of constraints is satisfied. If the constraints cannot be solved, or unique most-specific or most-general instantiations for the appropriate unbound type variables cannot be computed, then a static type error is reported.

The tests made as part of typechecking that involve the constraint solver are the following:

- whether one type T_1 is a subtype of another T_2 (directly expressible as the subtype constraint T_1 **subtypes** T_2 with no unbound type variables),
- whether one class $c[\overline{T}]$ conforms to a type T_2 (expressed as a conforms constraint $c[\overline{T}]$ **conforms** T_2 with no unbound type variables),
- whether a message $m[T_1,...,T_m]$ with argument types $T_1',...,T_n'$ is understood and what the most-specific result type α_r of the message is (expressed as the signature constraint **sig** $m[T_1,...,T_m](T_1',...,T_n'):\alpha_r$ where α_r is a fresh unbound type variable that should be minimized), and
- whether an application of a function of type T_λ to arguments of type $T_1,...,T_n$ is type-safe and what the most-specific result type α_r of the application is (expressed as the subtype constraint T_λ **subtypes** $\lambda(T_1,...,T_n):\alpha_r$ where α_r is a fresh unbound type variable that should be minimized).

To solve a system of constraints (initially a single constraint introduced as described above), the solver attempts to find or construct an *assumable* **subtypes**, **sig**, or **conforms** constraint that matches one of the constraints to be solved, and then removes the constraint from the set. If all constraints are removed, the system is satisfied and the typecheck passes. If the constraint solver is unable to construct an assumable constraint for one of the constraints, then the original constraint cannot be solved and the typecheck fails. Assumable constraints are found or located in several ways:

- An assumable constraint can be found directly in the context of assumed constraints from global monomorphic declarations or from **subtypes** or **sig** constraints in the enclosing **where** clause.
- An assumable constraint may be derived by instantiating a polymorphic declaration of the same kind. If the polymorphic declaration itself has constraints on its legal instantiation through a **where** clause of its own, then instantiated versions of these constraints are added to the system of constraints to be satisfied. Some of the declaration's type variables may be instantiated with fresh unbound type variables introduced internally by the typechecker; such variables are to be bound at later steps.
- Standard properties of signatures, subtyping, and conformance can be used to construct new assumable constraints from ones assumable through other techniques. For example, transitivity of subtyping and contravariant subtyping between function types allow building a subtype constraint from several others. Contravariance allows a new signature to be inferred from an existing one by making any of the argument types more specific and/or the result type more general. Two signatures that differ only in their result types can be used to infer a third whose result type is the **glb** of the result types of the first two. Other properties are specified formally in Section 3.2.

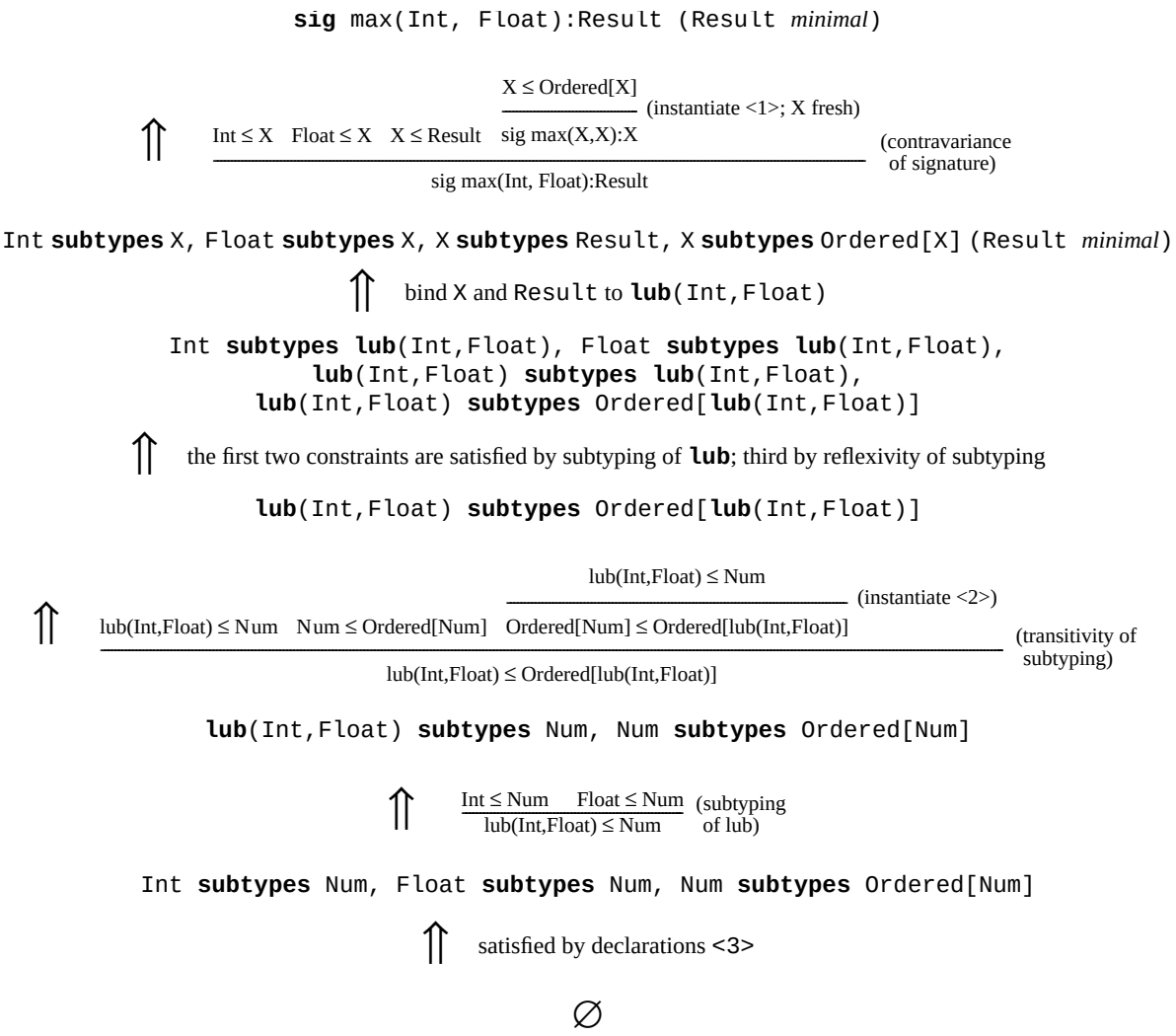

Figure 2: Example of Constraint Solving.
We use "≤" instead of "**subtypes**." The following declarations are referred to by numbers:
<1>: **forall** T **where** T **subtypes** Ordered[T]: **sig** max(T, T):T
<2>: **forall** S,T **where** S **subtypes** T: Ordered[T] **subtypes** Ordered[S]
<3>: Int **subtypes** Num; Float **subtypes** Num; Num **subtypes** Ordered[Num]

If matching a constraint with an unbound type variable against an assumable constraint (which has no unbound type variables), the type variable is bound to the corresponding type in the assumable constraint, replacing all other occurrences of the type variable in other constraints to be solved with the bound-to type. For the unbound type variable standing for the computed result of a message or application, the constraint solver is obligated to compute the unique most-specific binding, restricting the possible assumable constraints to those that meet this requirement; other internal type variables introduced during instantiation of polymorphic declarations may be instantiated to any legal type.

Figure 2 illustrates this process for typechecking the message `max(3,4.5)`. The typechecker infers that the message is type-correct and returns a value of type **lub**`(Int,Float)` (the type bound to the `Result` type variable).

The formal rules in Section 3 give a non-deterministic specification of our type system; here we described the deterministic algorithm we have implemented in our typechecker. The algorithm solves constraints by performing a search through an and-or proof tree. Or's arise when several possible choices exist for constructing an assumable constraint for a particular constraint to be solved; these choices stem from the many subtype and signature constraints that are declared or that can be constructed. A proof of any of the choices is sufficient as a proof of the constraint to be solved. And's arise from instantiations of polymorphic declarations that have **where** clauses listing multiple constraints. All of the constraints of the polymorphic declaration must be proved for the instantiation to be proved.

Conceptually, the and-or tree being searched is infinite. To avoid searching infinitely during typechecking, our implementation pursues only most-direct, most-general paths, using the constraint to be satisfied to construct an assumed constraint that causes a minimal set of new constraints to be introduced. However, this strategy is not always sufficient to guarantee termination of constraint solving. For example, the following declaration and expression causes unbounded constraint solving:

```
forall T, S where List[List[T]] subtypes List[List[S]]: List[T] subtypes List[S]
var x:List[Num] := ... -- r.h.s. of type List[Int]
```

To force constraint solving to be bounded, our implementation counts the number of steps that are taken in the search through the and-or tree, and reports as unsatisfied any constraints that take more than a fixed number of steps. This is one source of incompleteness of our deterministic algorithm with respect to the formal rules. Other sources include complicated uses of **lub** and **glb** types and in some cases recursive constraints like {`A[T]` **subtypes** `T`, `T` **subtypes** `B[T]`}. Incompleteness may cause the typechecker to report type errors in some cases which are legal according to the formal rules, reducing flexibility of the type system available to the programmer (however this doesn't happen for the Cecil code we have). This can be handled by improving the implementation algorithm without changing the formal type system.

Our future work includes formalizing the notion of most-direct, most-general searches and developing static well-foundedness restrictions on the bounds of polymorphic declarations to disallow declarations like the `List` declaration above, culminating with a proof of termination for our type system. Another direction is making the deterministic typechecking algorithm more complete.

2.6 Comparison with Other Type Systems

Since our system supports F-bounded polymorphism though subtype constraints, all of the standard "test" examples used in the literature to show the expressiveness of F-bounded polymorphism are supported in our system; these examples include the `Comparable` and `Ordered` types and binary methods presented earlier. (Multiple dispatching allows a different solution to the binary method problem, which offers a different semantics than the solution enabled by F-bounded polymorphism; our type system supports both solutions, allowing the programmers to choose the desired effect for different practical examples.) The following example, adapted from Thorup [Thorup 97], shows how mutually recursive subtype bounds can express the requirement that several interrelated types in a framework must be refined as a family to preserve type-safety.[*]

[*] Recall that Mini-Cecil uses an unsugared syntax that separates types and classes, for clarity of semantics. The full Cecil language supports syntactic sugars for defining both types and classes with a single set of declarations, requiring roughly half the text of the unsugared version.

```
-- The abstract framework over two types:
forall S,O where S subtypes Subject[S,O], O subtypes Observer[S,O]: {
  type Subject[S,O]
    sig observers(S):MutableList[O]
    sig notify_observers(S)
  abstract class SubjectClass[S,O]
    SubjectClass[S,O] conforms Subject[S,O]
    field observers(s@SubjectClass[S,O]:S):MutableList[O]
    method notify_observers(s@SubjectClass[S,O]:S):Void {
      do(observers(s), λ(e:O){ notify(o, s) }) }
  type Observer[S,O]
    sig notify(O, S):Void
}
-- An instantiation of the framework for a drawing subject and a drawing view observer, which is itself further extensible:
forall D,V where D subtypes Drawing[D,V], V subtypes View[D,V]: {
  type Drawing[D,V]
    Drawing[D,V] subtypes Subject[D,V]
    sig bitmap(D):BitMap
    sig set_pixel(D, Position, Color):Void
  concrete class DrawingClass[D,V]
    DrawingClass[D,V] conforms Drawing[D,V]
    DrawingClass[D,V] inherits SubjectClass[D,V]
    field bitmap(d@DrawingClass[D,V]:D):BitMap
    method set_pixel(d@DrawingClass[D,V]:D, p@Any:Position, c@Any:Color):Void {
      -- here observers(d) is known to contain (subtypes of) View which therefore have an update_pixel operation
      do(observers(d), λ(v:V){ update_pixel(v, p, c) }) }
  type View[D,V]
    View[D,V] subtypes Observer[D,V]
    sig update_pixel(V, Position, Color):Void
  concrete class ViewClass[D,V]
    ViewClass[D,V] conforms View[D,V]
    method notify(v@ViewClass[D,V]:V, d@Any:D):Void {
      -- here d is known to be a (subtype of) Drawing, and therefore has a bitmap
      plot(screen, bitmap(d)) }
    method set_pixel(d@ViewClass[D,V]:V, p@Any:Position, c@Any:Color):Void {
      plot_pixel(screen, p, c) }
}
```

Systems based on covariant method typing such as Eiffel [Meyer 92] or covariant type refinement such as Beta [Madsen & Møller-Pederse 89] and Thorup's proposed extension to Java allow this program, but also allow many other programs that are not type-safe; these languages either insert run-time typechecking or are unsafe. On the other hand, clients of our program must instantiate the various parameterized types and classes to use them, while the equivalent but unparameterized versions in Eiffel or Thorup's extension to Java may be manipulated directly. We plan to extend our language with syntactic and semantic support to define default instantiations of type parameters that clients can omit, which will make use of our framework as concise as in Eiffel or extended Java while preserving static type safety. Languages based on matching, such as the TOOPLE, TOIL, PolyTOIL, and LOOM series of languages [Bruce et al. 97], support a subset of F-bounded polymorphism that only allows the receiver argument to be constrained by itself, and as a result they cannot express these sorts of constraints over multiple types.

Our type system can express parameterized types where one instance of the parameterized type is a subtype of another instance, either in a covariant or contravariant manner. For example, the following declarations state that the read-only interface of an array is parameterized covariantly (since the type parameter only occurs in covariant positions in the signatures comprising the interface to a read-only array), while the read-write interface to an array is parameterized invariantly (since the type parameter appears in both covariant and contravariant positions in signatures in the full read-write interface).

```
forall T: {
  type Array[T]
    Array[T] subtypes Collection[T]
    forall S where S subtypes T: Array[S] subtypes Array[T]
    sig fetch(Array[T], Int):T
}
forall T: {
  type MutableArray[T]
  MutableArray[T] subtypes Array[T]
  sig store(MutableArray[T], Int, T):Void
}
var a1:MutableArray[Float] := ...;
store(a1, 0, 3.5);
var a2:Array[Num] := a1;
var n:Num := fetch(a2, 0);  -- n := 3.5
```

The following declarations extend the earlier `Comparable` and `Ordered` examples to reflect the fact that their type parameters are contravariant.

```
forall S, T where S subtypes T: Comparable[T] subtypes Comparable[S]
forall S, T where S subtypes T: Ordered[T] subtypes Ordered[S]
```

Few other languages can express this (for example, Pizza [Odersky & Wadler 97] and Theta [Day et al. 95, Liskov et al. 94] cannot). Strongtalk [Bracha & Griswold 93] and $ML_{\leq}$ [Bourdoncle & Merz 97] introduce specialized constructs to declare this relation; we do not. Array types in Java [Gosling et al. 96] and virtual types as in Thorup's proposal in effect treat all type parameters as covariant, which is a source of static unsoundness that necessitates run-time checking. In addition, we can limit the extent of contra- or covariance of a type parameter by adding additional constraints on the parameter introduced in the subtypes declaration, for example to limit contravariance of a function argument to types that are subtypes of `Num`.

Since our system supports Theta-style `where` clauses through signature constraints, all of the standard examples in support of `where` clauses are handled by our system. In addition to handling many of the examples used for F-bounded polymorphism (sometimes more concisely, sometimes less so), signature constraints can be used as part of dependent typing, enabling argument and result types to depend on the presence of other constraints. For example, signature constraints can be used to automatically infer appropriate interfaces to the `copy` operation on collections from whatever interfaces are already available for an underlying `copy_empty` operation (which produces an empty collection of the same representation as its argument, and is implemented for certain collection types but not others).

```
forall T, In, Out where In subtypes Collection[T],
                        sig copy_empty(In):Out,
                        sig add_last(Out, T):Void: {
  sig copy(In):Out;
  method copy(c@CollectionClass[T]:In):Out {
    var res:Out := copy_empty(c);
    do(c, λ(elem:T){ add_last(res, elem) });
    res }
}
... forall T: sig copy_empty(Array[T]):MutableArray[T] ...
... forall T: sig add_last(MutableArray[T], T):Void ...
... forall T: sig copy_empty(List[T]):MutableList[T] ...
... forall T: sig add_last(MutableList[T], T):Void ...
... forall T: sig copy_empty(Table[T]):MutableTable[T] ...
```

In this example, mutable and immutable `Array`s and `List`s can use this `copy` method, and clients will know that a value of the corresponding (mutable) type is returned. However, because `Table`s do not support `add_last`, they cannot reuse this `copy` function; presumably `Table`s provide their own `copy` function. Callers of `copy` do not provide the `T`, `In`, or `Out` type parameters; the system infers them from

the type of the argument to `copy` and from the available `copy_empty` and `add_last` signatures. Theta, by contrast, disallows methods with type parameters other than those inherited by the enclosing parameterized class declaration, and also requires all type parameters to be passed explicitly by clients.

3 Formal Description of the Type System

Typechecking a Mini-Cecil program proceeds in the following stages:

- the top-level type environment is created based on the program's top-level, mutually recursive declarations,
- client-side typechecking is performed, and
- implementation-side typechecking is performed.

Several of these steps in turn depend on checking whether instantiations are legal and checking systems of constraints for legality. Each of these five components is discussed in the following subsections.

This section serves as a non-deterministic specification of the behavior of the typechecker, not as a deterministic, executable algorithm for typechecking. We have implemented a deterministic algorithm; it is incomplete but is capable of handling all the Cecil code we have. Formalizing and improving this algorithm is future work.

Throughout the rules and explanation we use the overbar notation as follows. If $\overline{A} = A_1, ..., A_n$, $\overline{B} = B_1, ..., B_n$, C is a single element, and $\otimes$ is a binary operator, then $\overline{A} \otimes \overline{B} = A_1 \otimes B_1, ..., A_n \otimes B_n$ and $C \otimes \overline{B} = C \otimes B_1, ..., C \otimes B_n$.

3.1 Constructing the Top-Level Type Environment

All typechecking is performed in the context of a typechecking environment, Δ, which is a collection of declarations in scope.

Environment Δ ::= $\overline{CD}\ \overline{VD}$

For a given program $P = \overline{CD}\ \overline{VD}\ E$, its top-level environment Δ_P formed from the global mutually-recursive declarations as follows:

$$\Delta_P = \overline{CD}\ \overline{VD} \cup \Delta_{accessors}(P) \cup \Delta_{predefined}$$

$\Delta_{accessors}(P)$ contains, for each field declaration **forall** $\overline{\alpha}$ **where** $\overline{C}$: **field** $m[](v@c{:}T){:}T_r$ in P, two accessor method declarations **forall** $\overline{\alpha}$ **where** $\overline{C}$: **method** $m[](v@c{:}T){:}T_r$ and **forall** $\overline{\alpha}$ **where** $\overline{C}$: **method** `set_`$m[](v@c{:}T,\ v_{new}$`@Any:`T_r`):Void` (the bodies of these accessor pseudo-methods are ignored during typechecking). $\Delta_{predefined}$ contains the initial declarations needed by some parts of the semantics, including **type** `Void` and **abstract class** `Any`. We consider an n-argument function type $\lambda(T_1,...,T_n){:}T_r$ to denote a predefined type with n+1 parameters, `lambda/`$n[T_1,...,T_n,T_r]$, declared in $\Delta_{predefined}$; an analogous concrete class `lambdaClass/`$n[T_1,...,T_n,T_r]$ that conforms to `lambda/`$n[T_1,...,T_n,T_r]$ is also declared in $\Delta_{predefined}$. To encode the implicit contravariance of function types, $\Delta_{predefined}$ includes declarations like the following for all values of n:

forall $\alpha_1, \ldots, \alpha_n, \alpha_r, \beta_1, \ldots, \beta_n, \beta_r$ **where** β_1 **subtypes** $\alpha_1, \ldots,$ β_n **subtypes** α_n, α_r **subtypes** β_r:
`lambda/`$n[\alpha_1, \ldots, \alpha_n, \alpha_r]$ **subtypes** `lambda/`$n[\beta_1, \ldots, \beta_n, \beta_r]$

In Mini-Cecil, we disallow user-defined types or classes from subtyping from or conforming to, respectively, a `lambda` type; this ensures that anything of a `lambda` type is known statically to support the corresponding **apply** expression.* With this interpretation and restriction, function types and objects need no special treatment in the typechecking rules.

$$\frac{\Delta' = \Delta \cup D\,[\overline{\alpha} := \overline{T}] \qquad \Delta' \vdash C_1\,[\overline{\alpha} := \overline{T}]\ \textit{holds} \quad \ldots \quad \Delta' \vdash C_n\,[\overline{\alpha} := \overline{T}]\ \textit{holds}}{\Delta \vdash \textbf{forall}\ \overline{\alpha}\ \textbf{where}\ C_1, \ldots, C_n : D \rightarrow D\,[\overline{\alpha} := \overline{T}]}$$

Figure 4: Typechecking Instantiations

For typechecking each constrained declaration CD = **forall** α **where** C: D, a local environment Δ_{CD} is created as follows:

$$\Delta_{CD} = \Delta_P \cup \overline{C} \uplus \Delta_{vars}(CD)$$

Here constraints $\overline{C}$ are considered to be constrained declarations with empty contexts. $\Delta_{vars}(CD)$ is empty for all declarations except methods, when it contains a declaration **var** *v:T* for each formal *v@c:T*; for uniformity with variables we consider formals to be mutable in Mini-Cecil. The operation $\Delta_1 \uplus \Delta_2$ computes the union of Δ_1 and Δ_2, except that variable declarations in Δ_2 shadow declarations of variables with the same names in Δ_1.

The typechecker verifies that Δ_P contains a unique class or type declaration for every class or named type mentioned in the program, and that type variables are only used within constrained declarations that define them in their contexts. It is also verified that all syntactic occurrences of named types in the program are legal instantiations of the corresponding type declarations; such occurrences in the top-level variable declarations and expression are verified under Δ_P while occurrences in a constrained declaration CD are verified under Δ_{CD}.

3.2 Checking Constraints

Our typechecking rules depend on testing whether the following kinds of constraints hold, i.e., are provable:

subtype constraint	T_1 **subtypes** T_2 *holds* $\equiv$ $T_1 \leq_{sub} T_2$
signature constraint	**sig** $m[\overline{T}_p](\overline{T}_a):T_r$ *holds*
conforms constraint	$c[\overline{T}]$ **conforms** T_2 *holds* $\equiv$ $c[\overline{T}] \leq_{conf} T_2$

A constraint holds either if it is a legal instantiation of a declaration present in the environment, or if it is derived automatically from other constraints which hold. Figure 3 formalizes these rules. The rules for deriving subtype and signature constraints are mostly conventional; the rules involving **lub** and **glb** types illustrate how such types can be used. The rules for conformance constraints are similar to the subsumption and complementary rules for expressions.

3.3 Checking Instantiations

A constrained declaration CD can be instantiated to a declaration D by substituting types for any type variables in its context. Such instantiation is legal, written as $\Delta \vdash CD \rightarrow D$, if the constraints in the context of CD (if any) hold upon the substitution. Figure 4 gives the formal rule. (The notation $D\,[\overline{\alpha} := \overline{T}]$ replaces all references to α_i in D with T_i.) When checking whether a constraint holds, the instantiation being checked may be assumed, to support inductively defined constraints.

[*] Full Cecil treats function classes and types the same as other user-defined classes and types, without this restriction.

$$\frac{CD \in \Delta \quad \Delta \vdash CD \blacktriangleright D}{\Delta \vdash D \; holds}$$

$$\frac{\Delta \vdash T_1 \leq_{sub} \overline{T}}{\Delta \vdash T_1 \leq_{sub} \mathbf{glb}(\overline{T})}$$

$$\Delta \vdash T \leq_{sub} \texttt{Void}$$

$$\Delta \vdash T \leq_{sub} T$$

$$\frac{\Delta \vdash \mathbf{sig}\ m[\overline{T}_p](\overline{T}){:}T_r \; holds \quad \Delta \vdash \overline{T}' \leq_{sub} \overline{T} \quad \Delta \vdash T_r \leq_{sub} T_r'}{\Delta \vdash \mathbf{sig}\ m[\overline{T}_p](\overline{T}'){:}T_r' \; holds}$$

$$\frac{\Delta \vdash T_1 \leq_{sub} T_2 \quad \Delta \vdash T_2 \leq_{sub} T_3}{\Delta \vdash T_1 \leq_{sub} T_3}$$

$$\frac{\Delta \vdash \mathbf{sig}\ m[\overline{T}_p](T_1 \ldots T'_i \ldots T_n){:}T_r \; holds \quad \Delta \vdash \mathbf{sig}\ m[\overline{T}_p](T_1 \ldots T''_i \ldots T_n){:}T_r \; holds}{\Delta \vdash \mathbf{sig}\ m[\overline{T}_p](T_1 \ldots \mathbf{lub}(T'_i, T''_i) \ldots T_n){:}T_r \; holds}$$

$$\Delta \vdash T_1 \leq_{sub} \mathbf{lub}(T_1, \overline{T})$$

$$\Delta \vdash \mathbf{glb}(T_1, \overline{T}) \leq_{sub} T_1$$

$$\frac{\Delta \vdash \overline{T} \leq_{sub} T_1}{\Delta \vdash \mathbf{lub}(\overline{T}) \leq_{sub} T_1}$$

$$\frac{\Delta \vdash c[\overline{T}] \leq_{conf} T_1 \quad \Delta \vdash T_1 \leq_{sub} T_2}{\Delta \vdash c[\overline{T}] \leq_{conf} T_2}$$

$$\frac{\Delta \vdash c[\overline{T}_p] \leq_{conf} \overline{T}}{\Delta \vdash c[\overline{T}_p] \leq_{conf} \mathbf{glb}(\overline{T})}$$

Figure 3: Typechecking Constraints

3.4 Client-Side Typechecking

Client-side typechecking is performed on the program's top-level expression under Δ_P and on the body of each method declaration *CD* under Δ_{CD}, using the rules in Figure 5. The rules are mostly conventional, except that subtyping ($\leq_{sub}$), conformance ($\leq_{conf}$), and message typechecking rules rely on constraint testing, and class instantiation verifies that it is a legal instantiation. We also include the standard subsumption rule and the complementary rule that if several types can be inferred for an expression, the **glb** of them can also be inferred.

3.5 Implementation-Side Typechecking

Implementation-side typechecking ensures that all signatures are completely and consistently implemented. The criterion given in Figure 6 enumerates all legal instantiations of all signatures and tuples of conforming instantiations of concrete classes. For each combination it checks that a similar send at run-time will be successful and that there exists an instantiation of the target method which will be legal with respect to the arguments and the result type the typechecker may infer for this send. The method lookup process is abstracted in the *MethodLookup* helper function, derived from the underlying dynamic semantics and not formalized here. For field accessors, it is checked that the get- and set-accessor signatures provide a consistent view of the type of the fields' contents. (Note that this specification is not directly executable, as it enumerates an infinite number of possible instantiations. It is, however, a compact specification of the behavior of a bounded deterministic algorithm.)

$$\frac{B = \overline{VD}\ E \quad \Delta' = \Delta \uplus \overline{VD} \quad \Delta' \vdash E : T}{\Delta \vdash B : T}$$

$$\frac{\textbf{var}\ v : T \in \Delta}{\Delta \vdash v : T}$$

$$\frac{\textbf{var}\ v : T \in \Delta \quad \Delta \vdash E : T}{\Delta \vdash v := E : \texttt{Void}}$$

$$\frac{\Delta \vdash c \text{ names a } \textbf{concrete} \text{ class} \quad \Delta \vdash c[\overline{T}_p] \text{ is a legal instantiation} \quad \Delta \vdash c[\overline{T}_p] \leq_{conf} T}{\Delta \vdash \textbf{new}\ c[\overline{T}_p] : T}$$

$$\frac{\Delta \vdash \overline{E} : \overline{T} \quad \Delta \vdash \textbf{sig}\ m[\overline{T}_p](\overline{T}){:}T_r\ \textit{holds}}{\Delta \vdash m\ [\overline{T}_p](\overline{E}) : T_r}$$

$$\frac{\Delta' = \Delta \uplus \textbf{var}\ v_1{:}T_1 \uplus \ldots \uplus \textbf{var}\ v_n{:}T_n \quad \Delta' \vdash B : T_r}{\Delta \vdash \lambda(v_1{:}T_1, \ldots, v_n{:}T_n){:}T_r\ \{\ B\ \}\ :\ \lambda(T_1,\ldots,T_n){:}T_r}$$

$$\frac{\Delta \vdash E_\lambda : \lambda(\overline{T}){:}T_r \qquad \Delta \vdash \overline{E} : \overline{T}}{\Delta \vdash \textbf{apply}(E_\lambda\ \overline{E}) : T_r}$$

$$\frac{\Delta \vdash E_1 : T_1 \qquad \Delta \vdash E_2 : T_2}{\Delta \vdash E_1\ ;\ E_2 : T_2}$$

$$\frac{\Delta \vdash E : T_1 \quad \Delta \vdash T_1 \leq_{sub} T_2}{\Delta \vdash E : T_2}$$

$$\frac{\Delta \vdash E : \overline{T}}{\Delta \vdash E : \textbf{glb}(\overline{T})}$$

Figure 5: Typechecking Expressions

4 Experience with the Type System

We have implemented our type system in the Cecil language. We gained practical experience using the type system by applying it to the Vortex optimizing compiler system, a 100,000-line Cecil program. Vortex was originally developed with only dynamic typechecking; static type declarations were optional and unchecked, although developers were informally encouraged to write code that would potentially typecheck statically. In our experiment, we first ensured that all variables and interfaces had declared types, and then ran our typechecker. Of course, several thousand static type errors were reported in the first run, and we spent roughly a person-month of half-time effort in adjusting the type declarations to make the program statically typecheck. (In normal development, this time would have been spent incrementally as the program was developed, rather than all at once at the end of development.) A couple of static type errors remain, and around 350 explicit type casts were inserted; dynamic typechecking is used in these cases. Client-side typechecking is reasonably fast, taking about 7.5 minutes to typecheck all of Vortex from scratch on an UltraSPARC-1/170 workstation; implementation-side typechecking is missing several important but known optimizations [Chambers & Leavens 94, Chambers & Leavens 95] and so is slow.

For the most part, code (other than type declarations) did not need rewriting to meet the constraints of the static typechecker. F-bounded polymorphism was crucial to achieving this. In addition to describing "binary method" classes like `Comparable`, `PartiallyOrdered`, `Ordered`, and `Hashable`, F-bounded polymorphism was frequently used to properly type frameworks of interrelated classes. Such frameworks include a general directed graph framework and a refining partial order framework, the control flow graph

$\forall$ *signature declarations* $SigD_{poly} \in \Delta_P$
$\forall$ *tuples of concrete class declarations* $\overline{ClassD}_{poly} \in \Delta_P$ s.t. *length*$(\overline{ClassD}_{poly})$ = *number of args*$(SigD_{poly})$
$\forall$ *instantiations* $SigD$, $\overline{ClassD}$ s.t. $\Delta_P \vdash SigD_{poly} \blacktriangleright SigD$, $\Delta_P \vdash \overline{ClassD}_{poly} \blacktriangleright \overline{ClassD}$
  let $SigD$ = **sig** $m[\overline{T}_p](T_1, ..., T_n):T_r$, $ClassD_i$ = **concrete class** $c_i[\overline{T}_{ci}]$, $i = 1...n$ in
  $\Delta_P \vdash c_i[\overline{T}_{ci}] \leq_{conf} T_i$, $i = 1...n \Rightarrow$ *send-properly-implemented*$(m[\overline{T}_p], (c_1[\overline{T}_{c1}], ..., c_n[\overline{T}_{cn}]))$

send-properly-implemented$(m[\overline{T}_p], (c_1[\overline{T}_{c1}], ..., c_n[\overline{T}_{cn}])) =$
  if *MethodLookup*$(m[\overline{T}_p], (c_1, ..., c_n))$ fails then report error, otherwise
  let $MethD_{poly}$ = *MethodLookup*$(m[\overline{T}_p], (c_1, ..., c_n))$ in
  let T_{result} be the most specific type s.t. $\exists\ T_{conf\text{-}1}, ..., T_{conf\text{-}n}$ s.t.
    $\Delta_P \vdash c_i[\overline{T}_{ci}] \leq_{conf} T_{conf\text{-}i}$, $i = 1...n$ and $\Delta_P \vdash$ **sig** $m[\overline{T}_p](T_{conf\text{-}1}, ..., T_{conf\text{-}n}):T_{result}$ *holds* in
  exists-satisfactory-method-instantiation$(MethD_{poly}, \overline{T}_p, (c_1[\overline{T}_{c1}], ..., c_n[\overline{T}_{cn}]), T_{result})$ and
    (*MethD* is a get accessor method $\Rightarrow$
      set-accessor-signatures-consistent$(m[\overline{T}_p], (c_1[\overline{T}_{c1}], ..., c_n[\overline{T}_{cn}]), T_{result})$)

exists-satisfactory-method-instantiation$(MethD_{poly}, \overline{T}_p, (c_1[\overline{T}_{c1}], ..., c_n[\overline{T}_{cn}]), T_{result}) =$
  $\exists$ *instantiation* $MethD$ = **method** $m[\overline{T}_{inst\text{-}p}](v@c{:}T_{inst\text{-}a}):T_{inst\text{-}r}$ { B } s.t. $\Delta_P \vdash MethD_{poly} \blacktriangleright MethD$ and
    $\overline{T}_p = \overline{T}_{inst\text{-}p}$ and $\Delta_P \vdash c_i[\overline{T}_{ci}] \leq_{conf} T_{inst\text{-}ai}$ and $\Delta_P \vdash T_{inst\text{-}r} \leq_{sub} T_{result}$

set-accessor-signatures-consistent$(m[], (c[\overline{T}_c]), T_{result}) =$
  $\forall\ T_{conf}, T_{newval}, T_r\ .\ (\Delta_P \vdash c[\overline{T}_c] \leq_{conf} T_{conf}$ and $\Delta_P \vdash$ **sig** `set_`$m[](T_{conf}, T_{newval}):T_r$ *holds*$)$
    $\Rightarrow\ \Delta_P \vdash T_{newval} \leq_{sub} T_{result}$

Figure 6: Typechecking Implementations

node representation (with different possible instantiations of node representation), the program call graph representation (with mutually recursive classes for procedure nodes, call sites, and call edges), and intraprocedural and interprocedural analysis frameworks (with mutually recursive classes for analysis domains, transformation selections, and traversal state) [Chambers et al. 96]. Type parameters were used to parameterize over the types of mutable fields of classes, where the type of the field could become more specific in a subclass. 26 parameterized types used constrained subtype declarations to state that different instances of the parameterized type were subtypes if their instantiating parameter types were (i.e., had covariant type parameters), and 6 parameterized types were declared to have contravariant type parameters. Conditional subtyping and inheritance, applying only to certain instantiating types, as in the `Ordered Collection` example from section 2.4, occurred 25 times each. Automatic inference of parameter types was mandatory for practical use of the type system. 250 least-upper-bound type expressions and 85 greatest-lower-bound type expressions occur in the source code; more arise as part of constraint solving.

Surprisingly to us, signature bounds were less frequently used than F-bounds: in all, only 14 polymorphic declarations used signature bounds. The relative rarity could be due to signature bounds being a more recent addition to the Cecil type system than F-bounds and hence less familiar to the Vortex developers. It could also be because most F-bounding types have corresponding classes containing default implementation of many of the operations in the type. Syntactic sugar in Cecil makes using such F-bounded types easy, while similar syntactic support for naming and then referring to sets of signature constraints does not exist. Finally, since we were able and willing to reorganize existing type and class hierarchies, we did not benefit from one of the advantages of implicit structural subtyping using signature constraints over explicit by-name subtyping. The few cases of using signature constraints would be tedious to rewrite in terms of explicit types and subtyping, however.

A continuing weakness of the static typechecker was its inability to cope with the normal dynamically typed idioms for run-time type testing. For example, many class hierarchies defined predicate methods `is_X` for each externally meaningful notion *X*. Client code would test the outcome of the `is_X` tests and branch to code that invoked operations defined only on *X*'s.

```
sig is_X(AnyType):Bool
method is_X(s@Any:AnyType):Bool { False }
method is_X(s@SomeRep:X):Bool { True }
sig X_msg(X):Void
if(is_X(something), λ(){ ... X_msg(something) ... })
```

However, the static typechecker does not know that *X*-specific messages can be sent to the object, since the typechecker retains the original unnarrowed type for the tested value. A number of coding and/or language changes could be made to cope with this scenario, including introducing a `typecase`-like construct into the language or inserting casts by hand. In many situations, we modified the class hierarchies to define an `if_X` control structure that takes an argument function with one parameter of type *X*. All the classes that should be treated as the *X* abstraction provide an implementation of `if_X` that invokes its function argument, passing the receiver as an argument (which is known statically in these methods to be of type *X*); a default implementation of `if_X` does nothing (or invokes a second not-*X* argument function). Clients send the `if_X` message, passing a function with the *X*-specific operations as an argument.

```
sig if_X(AnyType, λ(X):Void):Void
method if_X(s@Any:AnyType, to_do:λ(X):Void):Void {}
method if_X(s@SomeRep:X, to_do:λ(X):Void):Void { apply(to_do, s) }
sig X_msg(X):Void
if_X(something, λ(x:X){ ... X_msg(x) ... })
```

This idiom is somewhat more cumbersome than the original code, but it requires no new language constructs and is statically type-safe. It also enables programming of interfaces with abstract properties that bear no relation to the underlying implementation hierarchy or even the type hierarchy; language-based constructs such as `typecase` that test the class of an object may expose too much of the internal implementation of an abstraction.

Because the program being typechecked had already been heavily tested, the typechecker only identified a few bugs in the program. It did point out some suspect code that could break after some otherwise legal and desirable future program evolution, and it helped us to reason about the interfaces to our more complicated subsystems. For a new interprocedural analysis framework implemented after the typechecker was stable and available to other Vortex developers, the typechecker did discover many errors statically that would otherwise have been encountered at run-time, and it did assist in carefully reasoning about the interfaces, all prior to defining any instantiations of the framework or running the code.

Some improvements to the type system would be helpful in making it more convenient to use and in encouraging highly polymorphic (and hence highly reusable) declarations. One key improvement would be to provide some automatic syntactic support for producing heavily-parameterized F-bounded classes, and for providing default instantiating types and specifying only differences from the defaults. Another improvement would be better syntactic support for abstracting and naming parameterizations and constraints, so that commonly occurring parameterization and constraint patterns do not need to be repeated. The type system might also benefit from more power, for example in supporting fully polymorphic values (variables and expressions whose types are polymorphic) in addition to its current, more-restrictive let-bound polymorphism. In a different direction, it may be reasonable to simplify the language by reunifying inheritance and subtyping; multimethods, F-bounded polymorphism, and signature constraints already handle most of the examples given as reasons for separating the two notions.

5 Related Work

We categorize related work on polymorphic type systems for object-oriented languages into several groups: languages based on F-bounded polymorphism and explicit subtyping, languages based on `SelfType` or matching, languages based on signature constraints and implicit structural subtyping, languages based on instantiation-time checking, and languages based on covariant redefinition. Our type system includes the core expressiveness of both F-bounded polymorphism (and its restrictions `SelfType` and matching) and signature constraints, provided uniformly over a wide range of declarations. Except where noted below, other languages based on these ideas support strict subsets of the expressiveness of our type system, although sometimes with more compact syntax. Also, the other languages do not support multi-methods, complete separation of inheritance from subtyping, and least-upper-bound and greatest-lower-bound type expressions, except where noted below.

5.1 Languages Based on F-Bounded Polymorphism

Pizza is an extension to Java based on F-bounded polymorphism [Odersky & Wadler 97]. Like our system, Pizza supports classes with mutually recursive bounds, crucial for supporting interrelated families of classes such as the `Subject`/`Observer` example from section 2.6. Also like our system, Pizza automatically infers instantiating type parameters of polymorphic methods and constructors, although the instantiating parameters must match the actual argument types exactly, which is more restrictive than our system which can infer appropriate supertypes of the argument types as in the `if` operation from section 2.2 and `min` and `max` operations from section 2.4. Pizza lacks signature constraints and the resulting implicit structural subtyping. Pizza does not support any subtyping between different instances of a parameterized type, such as the desirable and legal subtyping between different read-only interfaces to collection types as in section 2.6. Pizza also inherits several restrictions from its Java base, including that it does not allow contravariant method overriding. Pizza extends Java with first-class, lexically nested functions and with algebraic data types and pattern-matching. The authors justify introducing algebraic data types by claiming that classes allow new representations to be added easily but not new operations, while algebraic data types support the reverse. Multi-methods as in Cecil enable both new representations and new operations to be added easily, avoiding the need for new language constructs.

Bruce, Odersky, and Wadler [Bruce et al. 98] recently proposed to extend Pizza with special support for declaring families of mutually recursive classes. They argue that pure F-bounded polymorphism is too cumbersome for programmers to use in practice. We have not found pure F-bounded polymorphism to be untenable, however; the `Subject`/`Observer` example from section 2.6 illustrates our approach (albeit in an unsugared and hence verbose syntax). Our experience may be better than theirs because our multi-method framework encourages us to treat each argument and parameter symmetrically and uniformly, while their model is complicated by the asymmetry between the implicit receiver and the explicit arguments. Nevertheless, we too have been working on syntactic sugars that would make the more sophisticated uses of F-bounded polymorphism simpler.

Agesen, Freund, and Mitchell propose a similar extension to Java [Agesen et al. 97]. It differs from Pizza and from our system in being able to parameterize a class over its superclass. However, this feature cannot be typechecked once when the abstraction is declared, but instead must be rechecked at each instantiation.

Haskell's type classes can be viewed as a kind of F-bounded polymorphism [Wadler & Blott 89]. Haskell automatically infers the most-general parameterization and constraints on functions that take polymorphic arguments, as well as automatically inferring instantiations on calls to such functions; our system requires polymorphic methods to explicitly declare type variables and constraints over these variables. (In some cases, Haskell cannot unambiguously infer instantiations.) However, Haskell is not truly object-oriented, in

that after instantiation, no subtype polymorphism remains; values of different classes but a common supertype cannot be mixed together at run-time, preventing for instance lists of mixed integers and floats.

$ML_{\leq}$ is a powerful polymorphic object-oriented language supporting multi-methods [Bourdoncle & Merz 97]. Their language support subtyping directly, but treats inheritance as a separate syntactic sugar (which must follow the subtyping relation). Similarly to our system, they constrain type variables using sets of potentially recursive subtype constraints, they support inference of type parameters to methods, and they support least-upper-bound type expressions (although not greatest-lower-bound type expressions). They also support parameterization over type constructors, while in our system type constructors must be instantiated before use. They support explicit declarations of co- and contravariant type parameters of type constructors, while we use polymorphic subtype declarations to achieve more general effects. They only allow subtyping between types in the same type constructor "class," however, which for instance restricts subtyping to be between types with the same number of type parameters with the same variance properties, and they do not support other forms of constrained subtyping, conformance, or inheritance. Our type system supports multiple polymorphic signature declarations for the same message, while they allow only a single signature declaration per message. Their language is purely functional and side-effect-free.

5.2 Languages Based on `SelfType` or Matching

Some languages provide only restricted forms of F-bounded polymorphism. In TOOPLE [Bruce et al. 93] and Strongtalk [Bracha & Griswold 93], a special type `SelfType` is introduced, which can be used as the type of method arguments, results, or variables; roughly speaking, a class *C* with references to `SelfType` can be modeled with the F-bounded declaration

```
forall SelfType where SelfType subtypes C[SelfType]: class C[SelfType]
```

`SelfType` supports binary methods like `equal` and methods like `copy` that return values of exactly the same type as their receiver, but it does not support other kinds of F-bounded parameterization. Other languages provide a related notion called matching, which allows a kind of F-bounded polymorphism where a single type variable is bounded by a function of itself (but of no other type variables); languages including matching include PolyTOIL [Bruce et al. 95b] and LOOM [Bruce et al. 97]. The key advantage of `SelfType` and matching is convenient syntactic support for a common idiom, but it is less powerful than F-bounded polymorphism. For example, neither `SelfType` nor matching are powerful enough to support families of mutually dependent classes such as the `Subject`/`Observer` family. One of our main directions of current work is the development of syntactic support for making such mutually recursive families of classes easy to express, without restricting the underlying power of the polymorphic type system. Additionally, the LOOM language drops subtyping altogether in favor of matching, which costs it the ability to support run-time mixing of values of different classes but common supertypes, such as performing binary operations on the elements of a list of mixed integers and floats. `SelfType` and matching also are weaker than F-bounded polymorphism in that they force subclasses to continually track the more specific type; they cannot stop narrowing at some subclass and switch to normal subtyping below that point. For example, with F-bounded polymorphism, the parameterized `Ordered` type can have its type parameter "narrowed" and then fixed (say at `Ordered[Num]`), allowing subtypes of the fixed type (such as `Int` and `Float`) to be freely mixed. This open/closed distinction for recursive references to a type was noted previously by Eifrig *et al.* [Eifrig et al. 94].

5.3 Languages Based on Signature Constraints and Implicit Structural Subtyping

Some languages use collections of signatures to constrain polymorphism, where any type which supports the required signatures can instantiate the parameterized declaration. These systems can be viewed as treating the signature constraints as defining "protocol" types and then inferring a structural subtyping

relation over user-defined and protocol types. This inference is in contrast to the systems described earlier which require that the protocol types be declared explicitly, and that legal instantiations of the protocols be declared as explicit subtypes. Implicit structural subtyping can be more convenient, easier to understand, more adaptable to program evolution, and better suited to combining separately written code without change, while explicit by-name subtyping avoids inferring subtypings that ignore behavioral specifications, and may interact better with inheriting default implementations of protocol types. Neither is clearly better than the other; our polymorphic type system supports both easily. In addition, our underlying host language allows new supertypes to be added to previously declared types and classes, avoiding one limitation of explicit subtyping when adding new explicit protocol types and adapting previously written classes to conform to them.

Strongtalk is a type system for Smalltalk where programmers define protocol types explicitly, use protocols to declare the types of arguments, results, and variables, and let the system infer subtype and conformance relations between protocols and classes; like our system, subtyping and inheritance are separated. Precise details of the type system are not provided, but it appears that Strongtalk supports explicit parameterization (but without constrained polymorphism) for protocols and classes, a kind of parametric typing with dependent types and type inference for methods, least-upper-bound type expressions, and a form of `SelfType`. To avoid accidental subtyping, a class may be branded with one or more protocols. Like Cecil, type declarations and typechecking is optional in Strongtalk.

Interestingly, a later version of Strongtalk appears to have dropped inferred structural subtyping and brands in favor of explicit by-name subtyping [Bracha 96]. This later version also introduces the ability to declare that different instantiations of a parameterized type are subtype-related either co- or contravariantly with respect to its parameter types. Both Strongtalk systems are subsets of our type system.

The Theta language [Day et al. 95, Liskov et al. 94] and the similar proposed extension to Java [Myers et al. 97] support signature constraints called `where` clauses. Unlike our type system, only explicit type variables are supported, and clients must provide instantiations of all type variables when using a parameterized abstraction. No subtype relation holds between different instantiations of the same parameterized type, preventing idioms such as the covariantly related read-only collection interfaces.

Recursively constrained types are the heart of a very sophisticated type system [Eifrig et al. 95]. In this system, type variables and sets of constraints over them are automatically inferred by the system. Subtyping is inferred structurally, viewing objects as records and using standard record subtyping rules. Technically, the constraints on type variables are (mutually recursive) subtype constraints, but anonymous types may be introduced as part of the subtype constraints, providing a kind of signature constraint. Instead of instantiating polymorphic entities and inferring ground types for expressions, their system simply checks whether the inferred constraints over the whole program are satisfiable, without ever solving the constraints. For example, when computing the type of the result of a message, their system may return a partially constrained type variable, while our system must infer a unique, most-specific ground type. As a result, their system can typecheck programs our system cannot. On the other hand, because our system computes named types for all subexpressions, it can give simpler type error messages for incorrect programs; recursively constrained types can provide only the constraint system that was unsatisfiable as the error message, and this constraint system may be as large as the program source code itself. Their system limits syntactically where **lub** and **glb** subtype constraints can appear to ensure that such constraints can always be solved, while our system places no syntactic limits but may report a type error due to incompleteness of the particular deterministic algorithm used by the typechecker.

5.4 Languages Based on Instantiation-Time Checking

Some languages, including C++ [Stroustrup 91] and Modula-3 [SRC], dispense with specifying constraints on type variables entirely, relying instead on checking each instantiation separately. These languages are very flexible in what sort of parameterized declarations and clients can be written, as the only constraints that need be met are that the individual instantiations made in some program typecheck, and they are simple for programmers to use. (C++ also allows constant values as parameters in addition to types.) However, dropping explicit constraints on instantiating type variables loses the ability to check a parameterized declaration for type correctness once and for all separately from its (potentially unknown) clients, loses the specification benefit to programmers about how parameterized declarations should be used, and forces the source code of parameterized entities to be made available to clients in order for them to typecheck instantiations.

5.5 Languages Based on Covariant Redefinition

Some languages support bounded polymorphic classes through covariant redefinition of types or operations: a polymorphic class is defined as a regular class that has an "anchor" type member initialized to the upper bound of the type parameter, and instances are made by defining subclasses that redefine some anchor types to selected subtypes. Instances may themselves be further subclassed and their anchor types narrowed. Eiffel supports covariant overriding of methods and instance variables, and uses the `like` construct to refer to anchors [Meyer 92]; Eiffel also supports unbounded parameterized classes as well. Beta supports virtual patterns as anchor classes [Madsen & Møller-Pederse 89, Madsen et al. 93], and Thorup adapted this idea in his proposed virtual types extension to Java [Gosling et al. 96]. While all of these mechanisms seem natural to programmers in many cases and are syntactically concise, they suffer from a loss of static type safety. In contrast, our type system can directly support all of the standard examples used to justify such mechanisms (including binary methods, singly- and doubly-linked lists, and the `Subject`/`Observer` example), for instance using one or more mutually recursive F-bounded type parameters, without sacrificing static type safety. We are working on syntactic support for the general pattern of mutually recursive F-bounded type parameters, in hopes of achieving the same syntactic conciseness and programmer comprehensibility as well.

6 Conclusion

We have developed a polymorphic type system that uses systems of mutually recursive subtype and signature constraints to limit allowed type parameters. This constrained polymorphism is supported uniformly for all non-variable declarations in the system. This powerful, orthogonal treatment of polymorphism enables our type system to subsume F-bounded polymorphism, `SelfType`, matching, and Theta-style `where` clauses, and to directly express subtype relations over different instances of a polymorphic type. To use a polymorphic declaration, its type parameters need to be instantiated with ground argument types; in many cases the system can infer these type parameters automatically, and the system reports when it cannot. After instantiating a polymorphic declaration, run-time subtype polymorphism is still available, unlike Haskell and LOOM. A polymorphic declaration is typechecked only once, given the constraints on its type parameters, separately from any instantiations. Our type system supports other advanced features such as multi-methods, separate inheritance and subtyping, and first-class function objects.

We implemented this type system as part of the Cecil language, and used it to statically typecheck a 100,00-line Cecil program, the Vortex optimizing compiler system. Vortex was originally developed under a dynamically typed regime, and we wished the type system to be expressive enough to avoid forcing major rewriting of Vortex in order to statically typecheck. Our experience with this type system (unlike earlier,

less-expressive type systems implemented for Cecil) has been very good: code has needed little or no rewriting to adapt to the requirements of the type system. We encountered many situations in Vortex which required the advanced features of the type system to statically check successfully, such as families of mutually recursive classes and hierarchies of covariantly-parameterized read-only interfaces to collections; these experiences with real code rather than toy examples provide empirical justification for the added complexity of our approach.

Our type system is not a completed work, but rather an ongoing project. First, additional syntactic and semantic support for defining classes with many implicit type parameters for the different points of extensibility could improve the convenience, surface simplicity, and reusability of code statically typed using our system. Second, we are developing a deterministic description of the typechecking algorithm, adapted from the non-deterministic specification presented here. Third, we are investigating restrictions on constraints to ensure that they are inductively well-founded. Given these two components, we hope to prove termination of our algorithm, without recourse to an artificial limit on the number of steps allowed during constraint solving. Fourth, we are working on a proof of soundness of our type system. Our positive empirical experience using the type system strongly suggests that this point in the type system design space is worth pursuing.

Acknowledgments

We thank other members of the Cecil research project for their help in evaluating this and previous designs for Cecil's polymorphic type system. Dave Grove provided feedback on the presentation of this paper. This research is supported in part by an NSF grant (number CCR-9503741), an NSF Young Investigator Award (number CCR-9457767), and gifts from Sun Microsystems, IBM, Xerox PARC, Object Technology International, Edison Design Group, and Pure Software.

References

[Agesen et al. 97] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding Type Parameterization to the Java Language. In *Proceedings OOPSLA '97*, Atlanta, GA, October 1997.

[Bourdoncle & Merz 97] François Bourdoncle and Stephan Merz. Type Checking Higher-Order Polymorphic Multi-Methods. In *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 302–315.

[Bracha & Griswold 93] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a Production Environment. In *Proceedings OOPSLA '93*, pages 215–230, October 1993. Published as ACM SIGPLAN Notices, volume 28, number 10.

[Bracha 96] Gilad Bracha. The Strongtalk Type System for Smalltalk, 1996. OOPSLA '96 Workshop on Extending the Smalltalk Language, available from http://java.sun.com/people/gbracha/nwst.html.

[Bruce et al. 93] Kim B. Bruce, Jon Crabtree, Thomas P. Murtagh, Robert van Gent, Allyn Dimock, and Robert Muller. Safe and decidable type checking in an object-oriented language. In *Proceedings OOPSLA '93*, pages 29–46, October 1993. Published as ACM SIGPLAN Notices, volume 28, number 10.

[Bruce et al. 95a] Kim Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Objects Group, Gary T. Leavens, and Benjamin Pierce. On Binary Methods. *Theory and Practice of Object Systems*, 1(3):217–238, 1995.

[Bruce et al. 95b] Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyToil: A Type-Safe Polymorphic Object-Oriented Language. In *Proceedings ECOOP '95*, Aarhus, Denmark, August 1995. Springer-Verlag.

[Bruce et al. 97] Kim B. Bruce, Adrian Fiech, and Leaf Petersen. Subtyping is not a good "Match" for object-oriented languages. In *Proceedings ECOOP '97*. Springer-Verlag, June 1997.

[Bruce et al. 98] Kim B. Bruce, Martin Odersky, and Philip Wadler. A Statically Safe Alternative to Virtual Types. 1998. Submission to ECOOP '98, available from http://www.cs.williams.edu/ kim/README.html.

[Canning et al. 89] Peter Canning, William Cook, Walter Hill, John Mitchell, and Walter Olthoff. F-Bounded Polymorphism for Object-Oriented Programming. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280. ACM, 1989.

[Chambers & Leavens 94] Craig Chambers and Gary T. Leavens. Typechecking and Modules for Multi-Methods. In *Proceedings OOPSLA '94*, pages 1–15, Portland, OR, October 1994.

[Chambers & Leavens 95] Craig Chambers and Gary Leavens. Typechecking and Modules for Multi-Methods. *ACM Transactions on Programming Languages and Systems*, 17(9), November 1995.

[Chambers 92] Craig Chambers. Object-Oriented Multi-Methods in Cecil. In O. Lehrmann Madsen, editor, *Proceedings ECOOP '92*, LNCS 615, pages 33–56, Utrecht, The Netherlands, June 1992. Springer-Verlag.

[Chambers 93a] Craig Chambers. The Cecil Language: Specification and Rationale. Technical Report TR-93-03-05, Department of Computer Science and Engineering. University of Washington, March 1993.

[Chambers 93b] Craig Chambers. Predicate Classes. In O. Nierstrasz, editor, *Proceedings ECOOP '93*, LNCS 707, pages 268–296, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[Chambers et al. 96] Craig Chambers, Jeffrey Dean, and David Grove. Frameworks for Intra- and Interprocedural Dataflow Analysis. Technical Report UW-CSE-96-11-02, University of Washington, November 1996.

[Day et al. 95] Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Myers. Subtypes vs. Where Clauses: Constraining Parametric Polymorphism. In *OOPSLA'95 Conference Proceedings*, pages 156–168, Austin, TX, October 1995.

[Dean et al. 96] Jeffrey Dean, Greg DeFouw, Dave Grove, Vassily Litvinov, and Craig Chambers. Vortex: An Optimizing Compiler for Object-Oriented Languages. In *OOPSLA'96 Conference Proceedings*, San Jose, CA, October 1996.

[Eifrig et al. 94] Jonathan Eifrig, Scott Smith, Valery Trifonov, and Amy Zwarico. Application of OOP Type Theory: State, Decidability, Integration. In *Proceedings OOPSLA '94*, pages 16–30, Portland, OR, October 1994.

[Eifrig et al. 95] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound Polymorphic Type Inference for Objects. In *OOPSLA'95 Conference Proceedings*, pages 169–184, Austin, TX, October 1995.

[Goldberg & Robson 83] Adele Goldberg and David Robson. *Smalltalk-80: The Lanaguge and its Implementation*. Addision-Wesley, Reading, MA, 1983.

[Gosling et al. 96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, Reading, MA, 1996.

[Liskov et al. 94] Barbara Liskov, Dorthoy Curtis, Mark Day, Sanjay Ghemawhat, Robert Gruber, Paul Johnson, and Andrew C. Myers. Theta Reference Manual. Technical Report Programming Methodology Group Memo 88, MIT Laboratory for Computer Science, February 1994.

[Madsen & Møller-Pederse 89] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual Classes: A Powerful Mechanism in Object-Oriented Programming. In *Proceedings OOPSLA '89*, pages 397–406, October 1989. Published as ACM SIGPLAN Notices, volume 24, number 10.

[Madsen et al. 93] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Krysten Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley, 1993.

[Meyer 92] Bertrand Meyer. *Eiffel: the language*. Prentice-Hall, 1992.

[Myers et al. 97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized Types for Java. In *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 132–145.

[Odersky & Wadler 97] Martin Odersky and Philip Wadler. Pizza into Java: Translating Theory into Practice. In *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 146–159.

[SRC] DEC SRC Modula-3 Implementation. Digital Equipment Corporation Systems Research Center. http://www.research.digital.com/SRC/modula-3/html/home.html.

[Stroustrup 91] Bjarne Stroustrup. *The C++ Programming Language (second edition)*. Addision-Wesley, Reading, MA, 1991.

[Thorup 97] Kresten Krab Thorup. Genericity in Java with Virtual Types. In *Proceedings ECOOP '97*, Jyvaskyla, Finland, June 1997. Springer-Verlag.

[Wadler & Blott 89] Philip Wadler and Stephen Blott. How to Make *ad-hoc* Polymorphism Less *ad-hoc*. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 60–76, January 1989.