The Diesel Language

Specification and Rationale

Version 0.2

Craig Chambers

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

March, 2006

NOTE: This document is still under construction!

Table of Contents

1 Introduction. 5

2 Core 6

2.1 Programs and Files 6

2.2 Include Declarations 6

2.3 Variable Declarations 7

2.4 Classes and Objects 7

2.4.1 Kinds of Class Declarations 8

2.4.2 Inheritance 9

2.4.3 Object Instantiation 9

2.4.4 Predefined Objects and Classes 10

2.4.5 Class Extension Declarations 10

2.4.6 Synonym Declarations 11

2.5 Functions and Methods 11

2.5.1 Functions 11

2.5.2 Signatures 13

2.5.3 Methods 13

2.5.4 Code Method Bodies 17

2.5.5 Primitive Method Bodies 17

2.6 Fields 18

2.6.1 Read-Only vs. Mutable Fields 19

2.6.2 Per-Object vs. Shared Fields 19

2.6.3 Field Methods 20

2.6.4 Field Initialization 21

2.6.5 Field Overloading 23

2.7 Statements and Expressions 24

2.7.1 Assignment Statements 24

2.7.2 Literals 25

2.7.3 Variable References 25

2.7.4 Object Constructors 25

2.7.5 Vector Constructors 26

2.7.6 Closures 26

2.7.7 Message Sends 27

2.7.8 Parenthetical Subexpressions 28

2.8 Precedence Declarations 29

2.8.1 Previous Approaches 29

2.8.2 Precedence and Associativity Declarations in Diesel 30

2.9 Method Lookup 31
2.9.1 Philosophy 31
2.9.2 Semantics 32
2.9.3 Examples 33
2.9.4 Strengths and Limitations 35
2.9.5 Multiple Inheritance of Fields 36
2.9.6 Cyclic Inheritance 36
2.9.7 Method Invocation 36
2.10 Resends 37
2.11 Predicate Classes 39
2.11.1 Predicate Classes and Inheritance 40
2.11.2 Predicate Classes and Field Methods 44
2.12 Primitive Declarations 45
2.13 Pragmas 45
3 Static Types 46
3.1 Goals 46
3.2 Types and Signatures 47
3.3 Type Expressions 49
3.3.1 Named Types 49
3.3.2 Closure Types 50
3.3.3 Least-Upper-Bound Types 50
3.3.4 Greatest-Lower-Bound Types 50
3.4 Type Checking Messages 51
3.4.1 Checking Messages Against Signatures 51
3.4.2 Checking Signatures Against Method Implementations 52
3.4.3 Comparison with Other Type Systems 54
3.4.4 Type Checking Predicate Classes 55
3.5 Type Checking Expressions, Statements, and Declarations 57
3.6 Mixed Statically- and Dynamically-Typed Code 63
4 Parameterization and Bounded Parametric Polymorphism 66
4.1 Parameterized Declarations 66
4.2 Bounded Polymorphism and Type Constraints 68
4.3 Omitting the Explicit `forall` Clause: the Backquote Sugar 70
4.4 Polymorphism and Subtyping 72
4.5 F-bounded Polymorphism 74
4.6 Constraint Solving and Local Type Inference 76

4.7 Related Work 78
4.7.1 Languages Based on F-Bounded Polymorphism 78
4.7.2 Languages Based on `SelfType` or Matching 80
4.7.3 Languages Based on Signature Constraints and Implicit Structural Subtyping 80
4.7.4 Languages Based on Instantiation-Time Checking 81
4.7.5 Languages Based on Covariant Redefinition 82
4.7.6 Languages Offering Local Type Inference 82

5 Modules 84
5.1 Module Declarations 86
5.2 Privacy and Encapsulation 86
5.3 Qualified Names 89
5.4 Import Declarations 89
5.5 Extends Declarations 90
5.6 Friend Declarations 90
5.7 Module Extension Declarations 91
5.8 Function Call Overload Resolution 92

6 Related Work. 93

7 Conclusion 96

References 97

Appendix A Annotated Diesel Syntax 103
A.1 Grammar 103
A.2 Tokens 109
A.3 White Space 110

Index 111

1 Introduction

Diesel is a purely object-oriented language. All data are objects, and message passing is the only way to manipulate objects. Even instance variables are accessed solely using message passing. This purity offers the maximum benefit of object-oriented programming, allowing code to manipulate an object with no knowledge of (and hence no dependence on) its underlying representation or implementation. Diesel also allows the easy declaration of singleton objects with their own unique behavior. Diesel merges inheritance and instantiation.

Diesel is based on generic functions that support multiple dispatching based on the run-time classes of any subset of the function's arguments, rather than the more traditional single dispatching based solely on the run-time class of the receiver argument. Multiple dispatching unifies and generalizes receiver-oriented single dispatching and static overloading based on static argument types.

In part as a consequence of this multiple dispatching model, all functions, overriding methods, and even instance variables are declared separately from the classes on which they dispatch. This allows outside clients to easily extend existing classes with new functions, methods, and instance variables. Outside clients can also add new superclasses to existing classes.

Diesel supports first-class, lexically nested, anonymous function objects, called closures. Closures are commonly used to implement control structures entirely with normal Diesel code; Diesel has no built-in control structures.

Diesel supports predicate classes [Chambers 93b]. A predicate class is a "virtual" subclass of a regular class into which an instance of the regular class is automatically and dynamically classified whenever an associated predicate over the instance is true.

Diesel supports static typechecking. Diesel includes a constraint-based polymorphic type system, which allows expression of F-bounded polymorphism, "where" clauses, covariant and contravariant type parameters, and more. Diesel includes a limited form of type inference for instantiating type parameters.

Diesel includes a simple module system that allows programmers to manage the space of function and class names and to encapsulate the internals of a module from outside clients.

Diesel is a descendant of Cecil [Chambers 92b, Chambers 93a]. Diesel shares Cecil's multiple dispatching base and its constraint-based polymorphic type system. In contrast to Cecil, Diesel includes a module system, makes (generic) functions explicit, and unifies inheritance and subtyping.

The next section of this document describes the basic object, generic function, and message passing model. Section 3 presents the basics of the static type system. Section 4 explains parameterized types and the constraint-based polymorphic type system. Section 5 describes the module system. Section 6 discusses some related work, and section 7 concludes. Appendix A summarizes the complete syntax for Diesel.

2 Core

This section describes the core features of Diesel, including classes and objects, functions and methods, fields, and statements and expressions. It excludes many details about static types and typechecking (which are the subject of section 3), and ignores virtually all information about parameterized types (which are the subject of section 4) and modules (which are the subject of section 5).

2.1 Programs and Files

A Diesel program is made up of one or more files, starting with a root file. Each file is a sequence of (top-level) declarations and statements:*

```
program         ::= file_body
file_body       ::= { top_decl | stmt }
```

All the declarations introduced in a scope are visible throughout the scope, allowing forward references and mutually recursive declarations, without recourse to header files or forward declarations. Statements and any executable parts of declarations are executed in textual order.

The syntax of declarations is as follows:

```
top_decl        ::= include_decl
                |   static_decl
                |   dyn_decl
static_decl     ::= class_decl
                |   ext_class_decl
                |   predicate_decl
                |   disjoint_decl
                |   cover_decl
                |   divide_decl
                |   synonym_decl
                |   fun_decl
                |   method_decl
                |   signature_decl
                |   field_decl
                |   field_method_decl
                |   precedence_decl
                |   prim_decl
                |   pragma
dyn_decl        ::= let_decl
```

Since statements can be written at top-level, there is no need for a distinguished “main” function, nor is there any need for special module or class initialization code.

2.2 Include Declarations

Other files can be included in a program using `include` declarations:

```
include_decl    ::= "include" file_name {pragma} ";"
```

* Throughout this section, we ignore parameterization- and module-related constructs. These are the subject of Sections 4 and 5. Appendix A gives the complete syntax and lexical structure of the language and explains the grammar meta-notation.

```
file_name       ::= string
```

The included file should have the same syntax as the root file, i.e. a sequence of top-level declarations and statements. These additional declarations and statements are treated as being part of the same global scope as the root file. When the `include` declaration is executed, the included declarations and statements are executed. Included files themselves can include other files. A file may be included multiple times; the effect is that it is included only once, and its declarations and statements are executed only when the first `include` declaration for that file is executed. Include declarations can only appear at the top level, not in any nested scope.

The set of files comprising a Diesel program is the transitive closure of the `include` declarations, starting from the given root file. There is no need for any other extra-linguistic mechanism such as `Makefiles` to define a program.

2.3 Variable Declarations

Variable declarations have the following syntax:

```
let_decl        ::= "let" ["var"] name [type_decl] {pragma} ":=" expr ";"
type_decl       ::= ":" type
```

If the `var` annotation is used, the variable may be assigned a new value using an assignment statement. Otherwise, the variable binding is constant. (The contents of the variable may still be mutable.)

In most contexts, omitted type declarations default to `dynamic`, as described in section 3.6. However, for a constant variable binding within a dynamic context (i.e., a local variable, not a global variable), the type checker can easily and safely infer the type of the variable to be the same as the type of the iniitalizing expression, which allows many fewer explicit type declarations even in fully statically typed code.

All declarations in a scope are visible throughout the scope, including variables. However, a variable doesn't have a well-defined value until after its initializing expression has been executed. If a variable is read before its initializer has been executed, an "accessing uninitialized variable" run-time error is reported. In addition, in a dynamic scope, a compile-time error is reported if the variable is accessed before or during its declaration. This avoids potential misunderstandings about the meaning of apparently self-referential or mutually recursive initializers while still supporting a kind of `let*` [Steele 84] variable binding sequence. **verify that the implementation checks this way, vs. the old decl block way. could allow mutable vars w/o initializers, falling back on current error if read before assigned.**

2.4 Classes and Objects

The basic features of classes in Diesel are illustrated by the following declarations, which define a simple shape hierarchy. (Comments in Diesel either begin with "`--`" and extend to the end of the line or are bracketed between "`(--`" and "`--)`" and can be nested.)

```
abstract class Shape;
class Circle isa Shape;
```

```
class Rectangle isa Shape;
abstract class Rhombus isa Shape;
class Square isa Rectangle, Rhombus;
object UnitSquare isa Square;
```

The syntax of a class or object declaration is as follows:

```
class_decl     ::= class_kind name ["isa" class_refs] [field_inits]
                       {pragma} ";"
class_kind     ::= "abstract" "class"
                 | "class"
                 | "object"
                 | "prim" "class"
class_refs     ::= class_ref { "," class_ref }
class_ref      ::= name
```

(`name` is the token for regular identifiers beginning with a letter; see Appendix A.2 for more details on the lexical rules of Diesel.)

Note that a class declaration does not specify any members such as instance variables or methods. These are declared separately from classes, as discussed below. This deconstruction of the traditional monolithic class construct allows more flexible code organizations, for example allowing methods to be grouped in different ways than strictly by their enclosing class, and it allows clients to define new methods and instance variables for existing classes just as easily as the "base" functionality is. **mention/discuss "open classes" idea, if not somewhere around here already.**

Each class defines a corresponding type of the same name.

2.4.1 Kinds of Class Declarations

An abstract class is not allowed to be manipulated directly by programs as a value nor is it allowed to be instantiated. As a result, functions are not required to provide an implementing method case for them (such an omitted case is analogous to an abstract method in a traditional object-oriented language).

A regular class is a concrete class: it can be instantiated to produce first-class objects. Consequently, functions whose argument types admit arguments of the class's type must include an implementing method case for them. However, the class itself is not allowed to be manipulated directly by programs as a value.

An object declaration creates a first-class object. By being statically declared, it can have its own specialized type, functions, and method cases. **contrast with a let decl.** As with a concrete class, functions whose argument types admit the object must include implementing cases for it. An object declaration is good for defining one-of-a-kind "singleton" objects, without requiring a separate class along with programming idioms to ensure that only a single instance is ever created. Since it can be manipulated at run-time, all the fields of such an object should be initialized properly; field initializers are used for this purpose (and only make sense on an object declaration -- **FIX SYNTAX TO ENFORCE THIS**). Section 2.6.4 describes field initializers.

A "prim class" is one that is predefined by Diesel. **FINISH THIS. ADD PRIM FIELD PARTS.**

2.4.2 Inheritance

A class or named object can directly inherit from zero, one, or more other classes and/or named objects. We refer collectively to the immediate superclasses and superobjects as "parents," and the transitive closure of the parents as "ancestors;" "children" and "descendants" refer to the inverse relations. One effect of inheritance is on the reuse of implementation: a method case implemented for one class or object also applies to all its descendants; this is described in more detail in section 2.5. A second effect of inheritance is on subtyping: the type corresponding to a class or named object is a subtype of the types corresponding to each of its ancestors. If one type is a subtype of another, then objects of the subtype can be used wherever objects of the supertype are expected.

Inheritance in Diesel may be multiple, simply by listing more than one parent; any ambiguities among methods and/or fields defined on these parents will be reported to the programmer. Inheriting from the same ancestor more than once, either directly or indirectly, has no effect other than to place the ancestor in relation to other ancestors. A class or named object need not have any (explicit) parents; all classes and objects are considered to inherit from the predefined `any` object (see section 2.4.4). The inheritance graph must be acyclic (this is discussed more in section 2.9.6).

2.4.3 Object Instantiation

In addition to named object declarations, new objects can be created by evaluating *object constructor expressions*. For example:

```
... new Square ...   -- create a fresh instance of square when executed
```

The complete syntax of an object constructor expression is as follows:

```
object_expr    ::= "new" class_ref [field_inits]
```

The name of the instantiated class must be either a concrete class or a named object. The result is an anonymous object that inherits from the concrete class or named object. There is no separate "instance of" relation between an object and its class; instead, objects and classes just inherit from each other uniformly. The only difference between the object resulting from a named object declaration and the objects resulting from evaluating an object constructor expression is that the former have statically known names and associated named types; otherwise they are treated the same, e.g. inheritance of method cases works uniformly. An object constructor expression

new *C* **{** *inits* **}**

can be viewed as having the same effect as the object declaration

object <*anon*> **isa** *C* **{** *inits* **}**

In fact, classes themselves are really just named objects whose use is restricted. In this sense, Diesel can be viewed as an object-based (a.k.a. prototype-based or classless) rather than a class-based language, despite the use of the "class" keyword in its syntax. However, Diesel is not as fully prototype-based as some languages, such as Self [Ungar & Smith 87, Hölzle *et al.* 91a]: objects cannot inherit from run-time computed anonymous objects, nor can the inheritance of an object be changed at run-time.

Section 2.7.4 describes object constructor expressions in more detail.

2.4.4 Predefined Objects and Classes

Diesel includes several kinds of literal expressions, including integers, floats, characters, and (immutable) strings and described in section 2.7.2, each of which yields objects that are instances of a corresponding predefined concrete class. Similarly, `true` and `false` are predefined named objects which inherit from the predefined `bool` abstract class.

Diesel includes (immutable) vector constructor expressions, described in section 2.7.5, which create instances of a predefined concrete class.

Diesel includes closure constructor expressions, described in section 2.7.6, which create instances of a predefined concrete class. A closure object is a first-class, lexically nested, anonymous function. It is invoked by sending it the `eval` message, with additional actual arguments for each of the closure's formal arguments.

`any` is a predefined abstract class that is implicitly the ancestor of all other classes and objects. Consequently, behavior defined for `any` is inherited by all objects.

`void` is a predefined named object that can be used when there is no other useful value. `void` is the result of assignment statements, functions with no body, and the like, and it can be referenced as an expression in user methods that wish to return `void` explicitly. To preserve its intended use solely as a return value, `void` is not allowed as a superclass, nor a method specializer, nor can `void` be instantiated in an object constructor expression, and the `void` type cannot appear anywhere except as a return type.

The Diesel language does not define any additional behavior for these predefined objects and classes. Instead, the Diesel standard library specifies their behavior explicitly via regular functions and methods.

2.4.5 Class Extension Declarations

The inheritance structure and/or field initializers of a class or named object may be extended outside of the original declaration through a class extension declaration:

```
ext_class_decl ::= "extend" ext_class_kind name ["isa" class_refs]
                       [field_inits] {pragma} ";"
ext_class_kind ::= "class"
                 | "object"
```

Class extension declarations, in conjunction with function, method, and field declarations outside of classes, enable programmers to extend previously-existing classes and named objects. This ability can be important when reusing and integrating groups of classes implemented by other programmers. For example, predefined classes and named objects such as `prim_int`, `prim_i_string`, and `bool` are given additional behavior and ancestry through separate user code. Similarly, particular applications may need to add application-specific behavior to classes defined as part of other applications. For example, a text-processing application may add specialized tab-to-space conversion behavior to strings and other collections of characters defined in the standard library.

Most object-oriented languages do not allow programmers to add behavior to existing classes without modifying the source code of the existing classes, and completely disallow adding behavior to built-in classes like strings. Sather is a notable exception, allowing a new class to be defined which is a superclass of some existing classes [Omohundro 93]. **ADD REFS ABOUT OPEN CLASSES, VISITOR PATTERN, RETRO. ABSTRACTION.** Section 5 describes how modules can be used to localize the visibility of an extension to interested clients only.

2.4.6 Synonym Declarations

A new name for an existing class, named object, or type may be defined using a synonym declaration:

```
synonym_decl   ::= "synonym" name "=" type {pragma} ";"
```

A synonym is equivalent to the class, named object, or type to which it is defined, as opposed to a class declaration which introduces a new type distinct from any other type. It is primarily used to introduce a shorter name for a long type expression, e.g., one involving parameterized types.

A synonym can be used wherever the entity it is defined to be can be used. For example, if the synonym introduces a new name for a class, then the synonym can be inherited from, instantiated (if concrete), or used as a method specializer (as described in section 2.5.3).

A synonym can be defined in terms of other synonyms, as long as it is not defined directly or indirect in terms of itself.

2.5 Functions and Methods

2.5.1 Functions

Functions are the basic way that behavior is defined for objects. The following are some examples:*

```
abstract class Shape;
  fun draw(:Shape, :Display):void;
  fun draw(s:Shape):void { draw(s, Screen); }
  fun area(:Shape):num;
  fun move_to(s:Shape, new_center:Point):void { ... move s to new_center ... }
class Point;
  ... x and y field declarations ...
  fun +(p1:Point, p2:Point):Point { new_point(p1.x + p2.x, p1.y + p2.y) }
  fun new_point(x:num, y:num):Point { new Point { x := x, y := y } }
  fun new_origin():Point { new_point(0, 0) }
```

The syntax for function declarations is as follows:

```
fun_decl       ::= "fun" fun_name "(" [fun_formals] ")" [type_decl]
                       {pragma} fun_body
fun_name       ::= name | op_name
fun_formals    ::= fun_formal { "," fun_formal }
fun_formal     ::= [name] ":" type
                 |    name
```

* Indentation is semantically insignificant, but helpful for humans in grouping related declarations.

```
fun_body        ::= method_body | ";"
method_body     ::= "{" (body | prim_body) "}" [";"]
```

(`op_name` is the token for infix and prefix operators beginning with a punctuation symbol; see appendix A.2 for more details.)

A function declaration introduces a new function with the given name into the current scope, having either a "normal" name (a regular identifier, like `draw` and `move_to` above) or an "operator" name (like + above), and the given number of formal arguments. Functions with a normal name may have zero or more arguments (in which case the function is invoked using traditional function call syntax, described in subsection 2.7.7), while functions with an operator name must have either one formal (in which case the function is invoked as a prefix unary operator) or two formals (in which case the function is invoked as an infix binary operator; the relative precedence and associativity of infix operators can be specified explicitly using predecence declarations, described in section 2.8).

It is illegal to declare multiple functions with the same name and number of formal arguments in the same scope. Conversely, multiple functions with the same can be declared in the same scope as long as they have different numbers of formal arguments, as with the `draw` functions above. There is no static overloading of function names within a scope based on argument type, however.

Functions can be declared in a "static" scope, including at top-level and within a module declaration, but not in a "dynamic" scope such as within a function body or parenthetical subexpression. The effect of functions nested in dynamic contexts is achieved using closures, described in subsection 2.7.6. Similarly, functions themselves are not first-class objects, but the effect of a first-class function can be had via a closure whose body simply calls the function.
discuss issues with allowing nested functions & methods?

The body of a function can be omitted, leading to the Diesel equivalent of an "abstract method," as in the `area` and 2-argument `draw` functions above. Such a function must be overridden by one or more methods for each concrete argument combination, as described in section 3.4.2.

If present, the body of a function can be either a sequence of Diesel statements (as described in subsection 2.5.4) or a primitive written in an external language (as described in subsection 2.5.5).

If a formal argument is not used within the function's body (if present), its name can be omitted.

Any of the types of a function's arguments and/or its result type may be omitted. An omitted function argument or result type defaults to `dynamic`, which disables static type checking, as described in subsection 3.6. (Syntactically, each formal argument must have a name and/or a declared type; both cannot be omitted.)

The type of a function is captured by a *signature* that specifies the types of the formal arguments and the type of the result, as described in section 3.2. A function may be called on any actual argument objects that are subtypes of the corresponding argument types (where a type is a subtype of itself), and calls of the function will return an object that is a subtype of the result type.

The names of functions are in a name-space separate from the name-space of classes and variables. A function can have the same name as a variable or class without confusion.

2.5.2 Signatures

The type of a previously declared function can be refined using a signature declaration, whose syntax is as follows:

```
signature_decl ::= "signature" fun_ref "(" [fun_formals] ")" [type_decl]
                       {pragma} ";"
fun_ref        ::= name_fun_ref | op_fun_ref
name_fun_ref   ::= name
op_fun_ref     ::= op_name
```

A signature declaration augments the type of the given function (which must be declared separately) such that, if the function is invoked on actual argument types that are subtypes of the formals' types in the signature declaration, then the result of the invocation is also known to be a subtype of the signature's result type. The function's original type is still valid, as are any other signature declarations for that function. Rather than overriding the original function type in an incompatible way, signatures accumulate subtyping constraints on the possible result of an invocation; all such constraints must be satisfied by implementing methods, and all may be relied upon by invokers. (These are the usual rules for intersections of function types.)

For example, given a function like

```
fun copy(s:Shape):Shape;
```

signature declarations like

```
signature copy(r:Rectangle):Rectangle;
signature copy(r:Square):Square;
```

augment the original `(Shape):Shape` function type to also include `(Rectangle):Rectangle` and `(Square):Square`. So the `copy` function is known to return a `Shape`, and moreover, if the caller knows the argument is a `Rectangle`, the `copy` function is further known to return a `Rectangle`, and similarly if the caller further knows that the argument is a `Square`.

Typechecking rules for signature and method declarations, described in section 3.4.2, will cause signature declarations in practice to have formal argument types that are subtypes of the function's formal argument types and have a result type that is a subtype of the function's result type.

Parameterized functions, described in section 4, can specify uniform variations in a result type based on the argument types. In contrast, signature declarations allow more ad-hoc refinements in the type of a function to be specified.

2.5.3 Methods

A function's implementation can be overridden for particular combinations of argument objects using a method declaration. For example, the `draw` function can be overridden in various subclasses:

```
abstract class Shape;
  fun draw(:Shape, :Display):void;
class Circle isa Shape;
  method draw(c@Circle, d:Display):void { ... code for drawing a circle ... }
```

```
class Rectangle isa Shape;
  method draw(r@Rectangle, d:Display):void { ... code for drawing a rectangle ... }
  method draw(r@Rectangle, d@Xwindow):void {
    ... more specialized code for drawing a rectangle on an X window ... }
```

The syntax for method declarations is as follows:

```
method_decl    ::= "method" ["signature"] fun_ref "(" [meth_formals] ")"
                       [type_decl] {pragma} method_body

meth_formals   ::= meth_formal { "," meth_formal }

meth_formal    ::= [name] ":" type
                |  [name] "@" class_ref
                |  name
```

A method augments an existing function with the given name and number of arguments (methods do not introduce new functions). A method specifies a restricted combination of arguments for which its body is intended to be applicable; when the function is called on one of those argument combinations, the method's body is invoked in place of the function's body (if any). To express these restrictions, any of the formal arguments of a method may be *specialized*, by using the `@`*specializer* syntax instead of the `:`*type* syntax, where *specializer* is the name of a class or named object. For each of a method's specialized formals, the method is applicable only if the corresponding actual argument object is equal to or a descendant of the specializer class or named object. Any number of the formals of a method may be specialized, independently.

A formal's specializer should be a subtype of the function's corresponding declared argument type (if any). In contrast, the declared argument type (if any) of an unspecialized formal argument should be a supertype of the function's corresponding declared argument type (if any); this is the usual contravariant-argument rule for function subtyping and method overriding. As with signature declarations, the method's result type should be a subtype of the function's result type; this is the usual covariant-result rule for function subtyping and method overriding. (More precise rules, which also account for signatures, are in section 3.4.2.)

Many methods may augment the same function, as long as those methods have different combinations of argument specializers. One method may override another if it has more restrictive argument specializers. In general, when a function with a certain number of arguments is invoked, all the methods augmenting that function (plus the function itself, if it has a body) are examined to find those that are *applicable* to the actual argument objects. Of the applicable methods, the *single most-specific* method is chosen to invoke. One method is at least as specific as another if its specializers are pointwise at least as specific as the other's, i.e., for each argument position, either both method's formals are unspecialized, or the first method's formal is specialized and the other's either is unspecialized or is specialized to an object that's equal to or an ancestor of the first's specializer. A method is more specific than another if it is at least as specific as the other, and it is strictly more specific in at least one argument position. Note that this rule treats argument positions symmetrically; there are no "more important" arguments whose relative specificity takes precedence over others. If no methods are applicable, then a "message not understood" error is reported, while if multiple methods are applicable but none is uniquely most specific, then a "message ambiguous" error is reported. Static typechecking will warn about the potential for these

errors when examining function and method declarations, and run-time checking will test whether an error actually arises for any particular call.

For example, in the following code:

```
fun draw(:Shape, :Display):void { ... default drawing code ... }
method draw(c@Circle, d:Display):void { ... code for drawing a circle ... }
method draw(r@Rectangle, d:Display):void { ... code for drawing a rectangle ... }
method draw(s@Square, d:Display):void { ... code for drawing a square ... }
method draw(s@Square, d@Xwindow):void {
    ... more specialized code for drawing a square on an X window ... }
```

the four methods each are more specific than the function (because the methods specialize where the function does not), the two square methods are more specific than the rectangle method (because they have more specific specializers, uniformly), and the square-on-an-X-window method is more specific than the generic square method (because it specializes where the other does not, and the other arguments are at least as specific). The rectangle and circle methods are mutually unordered (neither is more specific than the other), but this is allowed as long as there are no objects that inherit from both `Circle` and `Rectangle`, since at most one of those methods will apply to any actual argument combination. The following method:

```
method draw(s:Shape, d@Xwindow):void {
    ... specialized code for drawing a shape on an X window ... }
```

is more specific than the function and less specific than the square-on-an-X-window method, but unordered with respect to the other methods. If an invocation of `draw` passed a `Circle` instance and an `Xwindow` instance, then both the circle method and the shape-on-an-X-window method would be applicable, but neither would be uniquely most-specific; such an invocation would then lead to a "message ambiguous" run-time error, and the static typechecker would warn about this possibility when examining the `draw` function and methods. As another example, if another concrete subclass of `Shape` were declared without defining a corresponding `draw` method:

```
class Triangle isa Shape;
```

then an invocation of `draw` that passed an instance of `Triangle` would not find any applicable methods and so would report a "message not understood" run-time error; the static typechecker would warn for this possibility when examining the concrete subclasses of `Shape` and the method implementations of `draw`. As a final example, if the following method were added:

```
method draw(r@Rectangle, d@Xwindow):void {
    ... specialized code for drawing a rectangle on an X window ... }
```

then this method would be unordered with respect to the following existing method, since different argument positions order the methods differently:

```
method draw(s@Square, d:Display):void { ... code for drawing a square ... }
```

Both methods would apply to an invocation of `draw` that passed an instance of `Square` and an instance of `Xwindow`, but neither would override the other. Fortunately, a third method also applies:

```
method draw(s@Square, d@Xwindow):void {
    ... more specialized code for drawing a square on an X window ... }
```

and this method overrides the first two, thereby resolving the ambiguity. This third method would be invoked at run time, and no run-time error would be reported, nor would any static warning be issued. As in this example, errors about ambiguous methods can be resolved by providing additional methods specialized on the ambiguously defined argument combinations.

More details about the rules for method lookup are given in section 2.9.

Diesel methods can emulate both traditional singly-dispatched methods (by specializing only the first argument) as well as true *multimethods* (by specializing on multiple arguments). Statically-overloaded functions and functions declared via certain kinds of pattern-matching also are subsumed by multimethods. Callers of a function cannot tell or depend on whether the function may be overridden by methods or on which argument positions the methods may specialize; these are internal implementation decisions that should not affect callers, and implementors of a function can always change these decisions without affecting any callers. For example, a given function can initially be implemented with a single unspecialized implementation and then later be extended or replaced with several specialized implementations, without affecting clients of the original function. In contrast, CLOS has a "congruent lambda list" rule that requires all methods in a particular generic function to specialize on the same argument positions.

A method declaration may include the `signature` keyword to implicitly generate a signature declaration each of whose argument types is the specializer (or, if unspecialized, the argument type) of the corresponding method formal argument, and whose result type is the method's result type. This provides value to clients of the function whenever the method declares more general argument types than the function (or other signatures augmenting the function) or a more specific result type. For example, the following method implies the earlier signature declaration:

```
method signature copy(r@Rectangle):Rectangle { ... }
```

A function declaration with a body is simply syntactic sugar for a function declaration without a body plus a method declaration with all unspecialized arguments.

The name of a formal may be omitted if it is not needed in the method's body. Unlike singly-dispatched languages, there is no implicit `self` formal in Diesel; all formals are listed explicitly.

Diesel's ability for methods to specialize on named objects supports something similar to CLOS's `eql` specializers. In CLOS, an argument to a multimethod in a generic function may be restricted to apply only to a particular *object* by annotating the argument specializer with the `eql` keyword. A Diesel method would simply specialize on the object, without additional language features. Diesel's mechanism differs from CLOS's in that in Diesel such a method also will apply to any descendants of the specializing object, while in CLOS the method will apply only for that object. Dylan, a descendant of CLOS, has a `singleton` specializer that is analogous to CLOS's `eql` specializer [Apple 92].

As mentioned in subsection 2.4.5, methods can be specialized on existing classes without needing to modify those existing classes. This facility, lacking in most object-oriented languages, can make reusing existing components easier since they can be adapted to new uses by adding functions, methods, fields, and even parents to them.

specify that method can be in a different scope than function being extended, as long as function is visible. specify that method can only be in a static scope, although allowing dynamically nested methods would be very cool.

2.5.4 Code Method Bodies

The body of a function or method can either be Diesel code or it can be code written in an external language. If Diesel code, the syntax is as follows:

```
body          ::= {dyn_decl | stmt} result
                  | empty                          return void
result        ::= normal_return                    return an expression
                  | non_local_rtn                  return from the lexically-enclosing method
normal_return ::= dyn_decl                         return void
                  | assignment [";"]               return void
                  | expr [";"]                     return result of expression
non_local_rtn ::= "^" [";"]                        do a non-local return, returning void
                  | "^" expr [";"]                 do a non-local return, returning a result
```

(The syntax and semantics of statements, assignments, and expressions is described in section 2.7.)

When invoked, a method evaluates its body in a new environment containing bindings for the method's formal parameters and nested in the method's lexically-enclosing environment. Formal parameters are treated as constant variable bindings and so are not assignable in the body.

If the the body is empty, the callee function or method returns the special `void` object (described in subsection 2.4.4) back to its caller.

Otherwise, the body evaluates its statements and then its final return clause. If the return clause is a declaration or an assignment, then the body returns `void` to the caller of the function or method. If the return clause is an expression, then the result of that expression (which might or might not be `void`) is returned to the caller. Otherwise, the return clause is a *non-local return*, prefixed with a `^` symbol. A non-local return is only useful inside a nested closure. It has the effect of returning its argument expression's result (or `void` if no argument expression is given) not to the caller of the closure (i.e., the sender of the `eval` message) but rather to the caller of the lexically enclosing function or method, just like a non-local return in Smalltalk-80 [Goldberg & Robson 83] and Self and similar to a `return` statement in C. **insert example.** A run-time error will result if a closure executes a non-local return after its lexically enclosing method has returned; first-class continuations are not supported.

2.5.5 Primitive Method Bodies

Alternatively, the body of a function or method may be written in an external language, such as C++ or the Diesel compiler's intermediate language. This is most useful for implementing basic primitive functionality, such as integer arithmetic, vector indexing, looping, and file I/O, that cannot be expressed in Diesel. The syntax of primitive bodies is as follows:

```
prim_body      ::= "prim" { language_binding }
language_binding::= language ":" code_string
                  | language "{" code_chars "}"
language       ::= name
```

```
code_string     ::= string
code_chars      ::= brace_balanced_chars        any characters, with balanced use of "{" and "}"
```

A primitive method's body is a list of (language name, implementation source code) pairs. The details of the protocol for writing code in another language inside a Diesel primitive method are implementation-specific. The UW Diesel implementation recognizes the `c_++`, `rtl`, and `wil` language names, for primitives written in C++ and the Vortex and Whirlwind compilers' internal intermediate languages, respectively. It is fairly straightforward to make calls to routines written in C++ from Diesel by defining a primitive method whose body is written in C++.

Looping primitive behavior is provided by the standard library's `loop` function specialized on the `closure` predefined class. The body of the loop function is a primitive that repeatedly invokes its argument closure until some closure performs a non-local return to break out of the loop. Diesel provides recursion but not looping, so looping is implemented as a primitive rather than recursively. Other languages such as Scheme [Rees & Clinger 86] avoid the need for such a primitive by relying instead on user-level tail recursion and implementation-provided tail-recursion elimination. However, tail-recursion elimination precludes complete source-level debugging [Chambers 92a, Hölzle *et al.* 92] and consequently is undesirable in general. The primitive `loop` method may be viewed as a simple tail-recursive method for which the implementation has been instructed to perform tail-recursion elimination.

2.6 Fields

Object state, such as instance variables and class variables, is supported in Diesel through *fields* and associated *accessor functions*. For example, to define a mutable instance variable `x` of type `T` for a particular class `C`, the programmer can declare a `field` of the following form:

```
var field x(:C):T;
```

This declaration allocates an internal "storage table" mapping each object of type `C` (or a subtype) to an object of type `T`.* It also defines two functions, named `x` and `set_x`, that provide the only way to access the internal table:

```
fun x(o:C):T { <return o.x> } -- the get accessor function
fun set_x(o:C, value:T):void { <o.x := value; return void> } -- the set accessor function
```

The *get accessor function* returns the object to which the argument object is mapped in the internal storage table, i.e., the current contents of its argument's instance variable. The *set accessor function* updates the internal storage table to map the first argument object to the second argument, i.e., it assigns to the first argument's instance variable, and then returns `void`. (Section 5 describes how these accessor methods can be encapsulated within the data abstraction implementation and protected from external manipulation.)

To illustrate, the following declarations specify some instance variables for part of the `Shape` hierarchy:

* A storage table is a semantically clean way to think about the per-object state for a field. An implementation typically spreads the space of the storage table across all the objects in the table, i.e., reserving space in each object to store the values of the fields defined for that object.

```
abstract class Shape;
   var field center(:Shape):Point; -- defines center(:Shape):Point and
                                   -- set_center(:Shape,:Point):void accessors
class Rectangle isa Shape;
   var field width(:Rectangle):num; -- defines width(:Rectangle):num and
                                    -- set_width(:Rectangle,:num):void accessors
   var field height(:Rectangle):num;-- defines height(:Rectangle):num and
                                    -- set_height(:Rectangle,:num):void accessors
```

Since a `Rectangle` is a subtype of `Shape`, every `Rectangle` object has storage for `center`, `width`, and `height`.

The syntax of field declarations is as follows:

```
field_decl     ::= ["shared"] ["var"] "field" name "(" fun_formal ")"
                       [type_decl] {pragma} field_body
field_body     ::= "{" body "}" [";"] | ";"
```

somewhere discuss possible extensions to fields with more than one argument.

ensure that the real parser expects fields and field methods to have exactly one formal, with the same syntax as functions and methods, respectively. handle fields w/ omitted arg types.

2.6.1 Read-Only vs. Mutable Fields

By default, a field is immutable: only the get accessor method is generated for it. To support updating the value of a field, the `var` prefix must be used with the field declaration. The presence of the `var` annotation triggers generation of the set accessor function. Immutable fields receive their values either as part of object creation or by an initializing expression associated with the field declaration, as described in section 2.6.4. Note that the contents of an immutable field can itself be mutable, but the binding of the field to its contents cannot change. (Global and local variables in Diesel similarly default to initialize-only semantics, with an explicit `var` annotation required to allow updating of the variable's value, as described in section 2.3.)

In general, we believe that it is beneficial to explicitly indicate when a field is mutable; to encourage this indication, immutable fields are the default. Programmers looking at code can more easily reason about the behavior of programs if they know that certain parts of the state of an object cannot be side-effected. Similarly, immutable fields support the construction of immutable "value" objects, such as complex numbers and points, that are easier to reason about.

Many languages, including Self and Eiffel, support distinguishing between assignable and constant variables, but few imperative languages support initialize-only instance variables. CLOS can define initialize-only variables in the sense that a slot can be initialized at object-creation time without a set accessor method being defined, but in CLOS the `slot-value` primitive function can always modify a slot even if the set accessor is not generated.

2.6.2 Per-Object vs. Shared Fields

By default, a field's storage table maintains a separate mapping for each object to its own field contents, i.e., each object inheriting a field declaration receives its own space to hold its value of the field. Alternatively, a field declaration may be prefixed with the `shared` keyword, in which

case the field stores a single value shared by all inheriting objects. A shared field thus acts like a class variable in Smalltalk or a `static` variable in Java. For example, the following declaration allocates space for a single value that is shared by all shapes:

```
shared var field default_color(:Shape):Color;
```

Shared fields create accessor functions just like regular fields. The accessors' implementation differs in that they access shared global memory rather than per-object memory. As with regular fields, shared fields can only be accessed by sending a message to an instance; there is no way to access a shared field directly.

2.6.3 Field Methods

A field declaration implicitly introduces one or two new functions, whose bodies have special implementations. As mentioned in section 2.5.3, a function with a body is just syntactic sugar for a function without a body plus a method containing the original function's body. A method with a special field accessor implementation is a *field method*. The field methods for a field can be declared explicitly without declaring any new functions, which is needed if the functions have already been declared. The syntax of a field method declaration is as follows:

```
field_method_decl ::= ["shared"] ["var"] "field" "method" ["signature"]
                          name_fun_ref "(" meth_formal ")" [type_decl]
                          {pragma} field_body
```

The following example illustrates how a subclass can implement a function via a field method declaration:

```
abstract class Shape;
    -- every Shape can report its width:
    fun width(:Shape):num;
class Rectangle isa Shape;
    -- Rectangles also allow their width to be changed:
    fun set_width(:Rectangle, :num):void;
    -- Rectangles implement width and set_width through storage:
    var field method width(@Rectangle):num;
class Circle isa Shape;
    -- Circles implement width through computation:
    method width(c@Circle):num { ... }
```

A field method is just like any other kind of method, and can override and be overridden just like any other kind of method. For example, if one class implements a function by computation, using a regular method, a subclass can choose to reimplement the function by storage, overriding the method with a field method. Conversely, if one class implements a function by storage, using a field or a field method, a subclass can choose to reimplement the function by computation, overriding the field method with a regular method. In the following code, the `AlignedRectangle` class can inherit from the `Polygon` class but override the `vertices` implementation to something more appropriate for axis-aligned rectangles:

```
class Polygon isa Shape;
    var field vertices(:Polygon):collection[Point];
```

```
    method draw(p@Polygon, d:Display):void {
        (-- draw the polygon on an output device, accessing vertices --) }

class AlignedRectangle isa Polygon;

    var field top(:AlignedRectangle);
    var field bottom(:AlignedRectangle);
    var field left(:AlignedRectangle);
    var field right(:AlignedRectangle);

    method vectices(r@AlignedRectangle):collection[Point] {
        -- assume++ is a binary operator, creating a new Point object
        [r.top    ++ r.left,  r.top    ++ r.right,
         r.bottom ++ r.right, r.bottom ++ r.left] }

    method set_vertices(r@AlignedRectangle, vs:collection[Point]):void {
        (-- set corners of rectangle from vs list, if possible --) }
```

Even if a field accessor method is overridden, it may remain accessible, since a resend from the overriding method may invoke the field accessor method. Consequently, the storage for the overridden field still exists. (The storage table model for the state of a field helps make this clear.) This makes it easy for a subclass to wrap a storage-based implementation of its superclass with additional computation. (Of course, implementations are free to optimize away the storage for a field in an object if it cannot be accessed, as with the `vertices` field in the `AlignedRectangle` class above.)

Accessing instance variables solely through automatically-generated accessor functions has a number of advantages over the traditional mechanism of direct variable access common in most object-oriented languages. Since instance variables can only be accessed through messages, all code becomes representation-independent to a certain degree. A subclass can alter the storage-vs-computation choices of its superclasses, without requiring changes in the superclass or its clients. Within a single class, programmers can change their minds about what is stored and what is computed without rewriting lots of client code. Syntactically, a simple message send that accesses an accessor function is just as concise as would be a variable access (using the `p.x` syntactic sugar, described in section 2.7.7), thus imposing no burden on the programmer for the extra expressiveness. Other object-oriented languages such as Self and Trellis have shown the advantages of accessing instance variables solely through special get and set accessor methods. CLOS enables get and/or set accessor methods to be defined automatically as part of the `defclass` form, but CLOS also provides a lower-level `slot-value` primitive that can read and write any slot directly. Dylan joins Self and Trellis in accessing instance variables solely through accessor methods. C#'s properties provide instance-variable-like access syntax to methods, but are less flexible since properties cannot be overridden by instance variables or vice versa, and are more verbose under coding conventions where all instance variables get property accessors.

2.6.4 Field Initialization

When an object is created (either by an object constructor expression or a named object declaration), an object-specific initial value may be specified for any of its non-shared fields. The syntax of field initializers is as follows:

```
field_inits    ::= "{" field_init { "," field_init } "}"
field_init     ::= name_fun_ref ["@" class_ref] ":=" expr
```

For example, the following function creates a new `Rectangle` and initializes its fields:

```
fun new_rectangle(c:Point, w:num, h:num):Rectangle {
   new Rectangle { center := c, width := w, height := h } }
```

For a field initialization of the form `name := expr`, the field to be initialized is found by performing a lookup akin to message lookup to find a field accessor method named `name`, starting with the object being created. Method lookup itself cannot be used directly, since the accessor method for the field to be initialized may have been overridden by a method of the same name. Instead, a form of lookup that ignores all regular methods is used. If this lookup succeeds in finding a single most-specific matching field accessor method, then that field is the one given an initial value. If no matching field or more than one matching field is found, then a "field initializer not understood" or an "ambiguous field initializer" error, respectively, is reported. The accessed field must be a non-shared field; if a field accessor method for a shared field is found, then a "initializing shared field" error is reported.

To resolve ambiguities and to initialize fields otherwise overridden by other fields, an extended name for the field of the form `name@C := expr` may be used instead. For these kind of initializers, lookup for a matching field begins with the class named `C` rather than the object being created; the object being created must inherit from `C`. Extended field names are analogous to a similar mechanism related to directed resends, described in section 2.10.

In addition, a field declaration can specify default initialization code, which has the same syntax as a method body. For example, if the height of a `Rectangle` should default to its width, the `height` field could be declared as follows:

```
var field height(r:Rectangle):num { r.width }
```

A field's default initializer are *not* evaluated unless and until needed when reading the field. If a field's get accessor method is invoked and the field's value for the argument object has not been set previously (either as part of object creation, by an earlier invocation of the field's set accessor method, or by an earlier invocation of the get accessor method), then the field's default initializer is evaluated (if no default initializer was specified, then the field accessor method reports an "accessing uninitialized field" error). The initializing expression may name the formal parameter of the field declaration, allowing the initial value of the field to be defined in terms of other state of the object of which the field is a part. It is not legal to read the value of a field during execution of the field's initializer; doing so will lead to an error or an infinite recursion. The result of the initializer is stored as the current value of the field (either for this argument object, if a non-shared field, or for the field as a whole, for a shared field), and returned as the result of the get accessor method.

By evaluating field initializers on demand rather than at declaration time, we avoid the need to specify some relatively arbitrary ordering over field declarations (as in Java and C#) or to resort to an unhelpful "unspecified" or "implementation-dependent" rule.

Evaluating a non-shared field's initializer expression repeatedly for each inheriting object seems to support common Diesel programming style. This corresponds to CLOS's `:initform` specifier. An alternative semantics would evaluate the field initializer at most once, and share the

resulting value across all objects that use the default. This semantics corresponds to CLOS's `:default-initargs` specifier. The difference in the semantics is exposed if the initializing expression evaluates to a new mutable object. In practice, it seems that each object wants its own mutable object rather than sharing the mutable object among all default-initialized objects. Moreover, the alternative semantics can be simulated by having a field's default initializer access a shared field holding the initial value.

It is not possible to override just the default initializer of a field or field method declaration, although this is sometimes desirable. The ability to override a field declaration's default initializer can be simulated by introducing a helper function computing the default initial value for its argument, and calling the helper in the field's initializer; the helper function can then be overridden to change the field's default.

2.6.5 Field Overloading

In a traditional language, different classes declared in the same scope can use the same name for their instance variables without conflict, e.g.:

```
class C {
   int x;
}

class D {
   string x;
}
```

However, when implementing this design in Diesel, the field declarations are expressed outside of their classes, and so are all in the same scope, e.g.:

```
class C;
   field x(:C):int;

class D;
   field x(:D):string;
```

Each field declaration with a given name generates a corresponding getter function declaration, all of which have the same name and so clash with each other, generating a duplicate function declaration error. (The same problem occurs when translating methods of traditional classes into Diesel functions outside of their classes, but the problem seems in practice to be more irritating for fields.)

There are several potential solutions to this clash. One is to encapsulate each of the classes in their own modules (described in section 5), which returns each field to being defined in its own scope. A lighter-weight solution is to treat the two fields as being in the same function, and use dynamic dispatching to resolve the clash. One way is simply to convert all but one of the `field` declarations into `field method signature` declarations, e.g.:

```
class C;
   field x(:C):int;

class D;
   field method signature x(@D):string;
```

The `signature` clause yields a signature that indicates that the function has an additional overloaded signature, potentially unrelated to its original signature, which clients can also call legally.

This approach favors one of the field declarations over the others. A more symmetric approach declares the function separately and then implements all instance variables as field method signatures on this function, e.g.:

```
fun x(:none):any;

class C;
   field method signature x(@C):int;

class D;
   field method signature x(@D):string;
```

To ensure that only the fields' signatures are useful, the separate function declaration uses a type that is a subtype of all possible function types with that number of arguments (as described in section 3.3, `none` is a subtype of all types, `any` is a supertype of all types, and functions obey standard contravariant subtyping rules).

2.7 Statements and Expressions

A statement is an assignment or an expression evaluated solely for its side-effects:

```
stmt            ::= assignment ";"
                |   expr ";"
```

An expression is either a literal, a reference to a variable or a named object, an object constructor expression, a vector constructor expression, a closure constructor expression, a message (written in one of several possible syntactic forms), a resend, or a parenthetical subexpression:

```
expr            ::= binop_expr
binop_expr      ::= binop_msg | unop_expr
unop_expr       ::= unop_msg | dot_expr
dot_expr        ::= dot_msg | simple_expr
simple_expr     ::= literal
                |   var_expr
                |   vector_expr
                |   closure_expr
                |   object_expr
                |   message
                |   resend
                |   paren_expr
```

All of these constructs are described below, except for resends which are described in section 2.10.

2.7.1 Assignment Statements

Assignment statements have the following syntax:

```
assignment      ::= var_ref ":=" expr          assignment to a variable
                |   assign_msg                 assignment-like syntax for messages
var_ref         ::= name
```

If the left-hand-side is a simple name, then the closest lexically-enclosing binding of the name is located and changed to refer to the result of evaluating the right-hand-side expression. It is an error to try to assign to an object, a formal parameter, or a variable declared without the `var` keyword.

If the left-hand-side has the syntax of a message, then the assignment statement is really syntactic sugar for a message send, as described in section 2.7.7.

2.7.2 Literals

Diesel literal constants include integers, floating-point numbers, characters, and strings:

```
literal         ::= integer
                |   single_float
                |   double_float
                |   character
                |   string
```

An integer literal is an instance of the predefined `prim_int` class, and has an allowed range between 0 and an implementation-dependent maximum. Negative integers are computed e.g. by negating a positive integer. (In the UW Diesel implementation, the standard library defines `max_int`, the largest possible instance of `prim_int`. Arbitrary-precision integers are also supported, through a separate user-defined library class.)

Single- and double-precision foating-point literals are instances of the predefined `prim_single_float` and `prim_double_float` classes, respectively.

A character literal is an instance of the predefined `prim_char` class, and supports ASCII character codes in the range [0..255]. (In the UW Diesel implementation, Unicode characters are also supported, through a separate user-defined library class.)

A string literal is an instance of the predefined `prim_i_string` class. A string stores a possibly-empty sequence of `prim_char` objects.

The value of a literal object is immutable, and is accessible only through primitives (described in section 2.5.5). (In the UW Diesel implementation, mutable strings are also supported, through a separate user-defined library class.)

2.7.3 Variable References

A variable or named object is referenced simply by naming the variable or object:

```
var_expr        ::= var_ref
```

The names of classes, objects, and variables are in the same name-space. Lexical scoping is used to locate the closest lexically-enclosing declaration in this name-space. If the declaration is a class, an error is reported. Otherwise, the named object or the current contents of the variable is returned.

2.7.4 Object Constructors

New objects are created either through object declarations (as described in section 2.4) or by evaluating object constructor expressions (as discussed in section 2.4.3). The syntax of an object constructor expression is as follows:

```
object_expr    ::= "new" class_ref [field_inits]
```

An object constructor expression creates a new anonymous object that directly inherits from the named class. Any field initializers are evaluated to set the initial values of the referenced fields, as described in section 2.6.4.

2.7.5 Vector Constructors

A vector constructor expression is written as follows:

```
vector_expr    ::= "[" [":" type ":"] [exprs] "]"
exprs          ::= expr { "," expr }
```

The result of evaluating a vector constructor expression is a new instance of the predefined `prim_i_vector[`*T*`]` object that is initialized with the corresponding elements. The elements of the vector are immutable, and are accessible only through primitives (described in section 2.5.5). (In the UW Diesel implementation, mutable vectors are also supported, through a separate user-defined library class.) The type of the vector's elements can be specified explicitly, or inferred as the least-upper-bound of the types of the initial elements, as described in section 3.5.

2.7.6 Closures

A closure is an anonymous, lexically nestable, first-class function object. The syntax of a closure constructor expression is as follows:

```
closure_expr   ::= [ "&" "(" [closure_formals] ")" [type_decl] ] "{" body "}"
closure_formals::= closure_formal { "," closure_formal }
closure_formal ::= [name] ":" type
                 |   name
```

This syntax is like that of a function declaration, except that the `fun` keyword and message name are replaced with the `&` symbol (intended to be suggestive of the λ symbol). If the closure takes no arguments, then the `&()` prefix may be omitted. When evaluated, a closure constructor produces two things:

- a new instance of the predefined `closure` class, which is returned as the result of the closure constructor expression, and
- a method in the predefined `eval` function whose anonymous first argument is specialized on the newly-created closure object and whose remaining arguments are those listed as formal parameters in the closure constructor expression.

The body of a closure's `eval` method is lexically-scoped within the scope that was active when the closure was created. Closures may be invoked after their lexically-enclosing scopes have returned.*

A closure's result type annotation can be omitted, in which case it is inferred to be the same as the type of the result expression in the body, or `none` if the closure ends in a non-local return.

insert cross-ref to non-local returns in 2.5.4

* In the current UW Diesel implementation, there are some caveats to the use of such non-LIFO closures. See the system documentation for additional details.

All control structures in Diesel are implemented at user level using messages and closures, with the sole exception of the `loop` primitive method described in section 2.5.5. Additionally, closures can be used to achieve much the same effect as exceptions, so exceptions are omitted from the Diesel language. **WRITE MORE?**

EXPLAIN HOW EVAL AS MESSAGE MORE FLEXIBLE, BUT MORE VERBOSE, THAN BUILT-IN INVOKE PRIMITIVE.

EXPLAIN TRADEOFFS BETWEEN CAPABILITIES OF GENERIC FUNCTIONS AND CLOSURES.

document closure types as classes, which can be subclassed by other than built-in closures.

2.7.7 Message Sends

The syntax of message sends includes the following:

```
message         ::= name_fun_ref "(" [exprs] ")"
unop_msg        ::= op_fun_ref unop_expr
binop_msg       ::= binop_expr op_fun_ref binop_expr
```

A message can be written in one of three forms:

- named prefix form, with the name of the message followed by a parenthesized list of expressions,*
- unary operator prefix form, with the message name listed before the argument expression, or
- infix form, with the message name in between a pair of argument subexpressions.

Normally, a message whose name begins with a letter is written in named prefix form, while a message whose name begins with a punctuation symbol is written in unary prefix form or in infix form.† To invoke a named message as an operator, or to invoke an operator as a named message, the name of the message is prefixed with an underscore (the leading underscore is not considered part of the message name). For example, the following two expressions both send the + message to 3 and 4:

```
3 + 4
_+(3, 4)
```

and the following two expressions both send the `bit_and` message to 3 and 4:

```
bit_and(3, 4)
3 _bit_and 4
```

The relative precedence and associativity of infix messages is specified through precedence declarations, described in section 2.8. (The relative precedence and associativity of other syntactic forms of messages are already completely defined by the grammar.)

Syntactic sugar exists for several common forms of messages. Dot notation allows the first argument of the message to be written first:

* All arguments to the message must be listed explicitly; there is no implicit `self` argument.

† Named prefix form is always used for function and method declarations.

```
dot_msg        ::= dot_expr "." name_fun_ref ["(" [exprs] ")"]
```

If the message takes only one argument, the trailing parentheses can be omitted. Consequently, the following three expressions all send the `x` message to `p`:

```
x(p)
p.x()
p.x
```

The following two expressions both send the `bit_and` message to 3 and 4:

```
bit_and(3, 4)
3.bit_and(4)
```

This syntax may suggest that the first argument is more important than the others, but in fact the semantics is still that all arguments are treated uniformly, and any subset of the arguments might be dispatched at method-lookup time.

Other syntactic sugars support message sends written like assignments. Any message can appear on the left-hand-side of an assignment statement:

```
assign_msg     ::= lvalue_msg ":=" expr
lvalue_msg     ::= message
                 |   dot_msg
                 |   unop_msg
                 |   binop_msg
```

In each of these cases, the name of the message sent to carry out the "assignment" is `set_` followed by the name of the message in the `lvalue_msg` expression, and the arguments to the real message are the arguments of the `lvalue_msg` expression followed by the expression on the right-hand-side of the "assignment." So the following three expressions are all equivalent:

```
set_foo(p, q, r);
foo(p, q) := r;
p.foo(q) := r;
```

as are the following two expressions:

```
set_top(rectangle, x);
rectangle.top := x;       -- frequently used for set accessor methods
```

as are the following two expressions:

```
set_!(v, i, x);
v!i := x;
```

Note that these syntactic sugars are assignments in syntax only. Semantically, they are all messages.

The semantics of method lookup are described in section 2.9. Resends, a special kind of message send, are described in section 2.10.

2.7.8 Parenthetical Subexpressions

A parenthesized subexpression has the same syntax as the body of a function, method, or closure:

```
paren_expr     ::= "(" body ")"
```

A parenthetical subexpression introduces a new nested scope and may contain statements and local declarations.

2.8 Precedence Declarations

Diesel programmers can define their own infix binary operators. Parsing expressions with several infix operators becomes problematic, however, since the precedence and associativity of the infix operators needs to be known to parse unambiguously. For example, in the following Diesel expression

```
foo ++ bar *&&! baz *&&! qux _max blop
```

the relative precedences of the `++`, `*&&!`, and `_max` infix operators is needed, as is the associativity of the `*&&!` infix operator. For a more familiar example, we'd like the following Diesel expression (where `**` represents exponentiation)

```
x + y * z ** e ** f * q
```

to parse using standard mathematical rules, as if it were parenthesized as follows:

```
x + ((y * (z ** (e ** f))) * q)
```

2.8.1 Previous Approaches

Most languages restrict infix operators to a fixed set, with a fixed set of precedences and associativities. This is not appropriate for Diesel, since we'd like the set of infix messages to be user-extensible.

Smalltalk defines all infix operators to be of equal precedence and left-associative. While simple, this rule differs from standard mathematical rules, sometimes leading to hard-to-find bugs. For example, in Smalltalk, the expression `3 + 4 * 5` returns 35, not 23.

Self attempts to rectify this problem with Smalltalk by specifying the relative precedence of infix operators to be undefined, requiring programmers to explicitly parenthesize their code. This avoids problems with Smalltalk's approach, but leads to many unsightly parentheses. For example, the parentheses in the following Self code are all required:

```
(x <= y) && (y <= (z + 1))
```

Self makes an exception for the case where the same binary operator is used in series, treating that case as left-associative. For example, the expression

```
x + y + z
```

parses as expected in Self. Even so, the expression

```
x ** y ** z
```

would parse "backwards" in Self, if `**` were defined. (Self uses `power:` for exponentiation, perhaps to avoid problems like this.) Also, expressions like

```
x + y - z
```

are illegal in Self, requiring explicit parenthesization.

Standard ML [Milner *et al.* 90] allows any operator to be declared prefix (called "nonfix" in SML) or infix, and infix operators can be declared left- or right-associative. Infix declarations also specify a precedence level, which is an integer from 0 (loosest binding) to 9 (tightest binding), with 0 being the default. For example, the following SML declarations are standard:

```
infix 7 *, /, div, mod;
infix 6 +, -;
```

```
infix 4 = <> < > <= >=;
infix 3 :=;
nonfix ~;
```

SML also provides special syntax to use an infix operator as a prefix operator, and vice versa.

A fixity declaration can appear wherever any other declaration can appear, and affect any parsing of expressions while the fixity declaration is in scope. Fixity declarations can be spread throughout a program, and multiple declarations can add independent operators to the same precedence level. Fixity declarations in one scope override any fixity declarations of the same operator from enclosing scopes.

One disadvantage of SML's approach is that is supports only 10 levels of precedence. It is not possible to add a new operator that is higher precedence than some operator already defined at level 9, nor is it possible to squeeze a new operator in between operators at adjacent levels. Finally, all operators at one level bind tighter than all operators at lower levels, even if the programmer might have preferred that expressions mixing operators from completely different applications be explicitly parenthesized, for readability.

2.8.2 Precedence and Associativity Declarations in Diesel

Diesel allows the precedence and associativity of infix operators to be specified by programmers through precedence declarations. The syntax of these declarations is as follows:

```
prec_decl      ::= "precedence" op_names [associativity] {precedence} ";"
associativity  ::= "left_associative" | "right_associative" | "non_associative"
precedence     ::= "below" op_names | "above" op_names | "with" op_names
op_names       ::= op_name { "," op_name }
```

For example, the following declarations might appear as part of the standard Diesel library:

```
precedence ** right_associative;   -- exponentiation
precedence *, / left_associative below ** above +;
precedence +, - left_associative below * above =;
precedence =, !=, <, <=, >=, > non_associative below * above;
precedence & left_associative below = above |;
precedence | left_associative below &;
precedence % with *;
precedence ! left_associative above =;   -- array indexing
```

By default, an infix operator has its own unique precedence, unrelated to the precedence of any other infix operator, and is non-associative. Expressions mixing operators of unrelated precedences or multiple sequential occurrences of an operator that is non-associative must be explicitly parenthesized.

The effect of a precedence declaration is to declare the relationship of the precedences of several binary operators and/or to specify the associativity of a binary operator. Like SML, the information provided by a precedence declaration is used during the scope of the declaration, and declarations of the same operator at one scope override any from an enclosing scope. Two precedence declarations cannot define the precedence of the same operator in the same scope.

A precedence declaration of the form

precedence *bin-op*$_1$, ..., *bin-op*$_n$
associativity
below *bin-op*$_{B1}$, ..., *bin-op*$_{Bn}$

above *bin-op*$_{A1}$**, ...,** *bin-op*$_{An}$
with *bin-op*$_{W1}$**, ...,** *bin-op*$_{Wn}$**;**

declares that all the *bin-op*$_i$ belong to the same precedence group, and that this group is less tightly binding than the precedence groups of any of the *bin-op*$_{Bi}$ and more tightly binding than those of the *bin-op*$_{Ai}$. If any *bin-op*$_{Wi}$ are provided, then the *bin-op*$_i$ belong to the same precedence group as the *bin-op*$_{Wi}$; all the *bin-op*$_{Wi}$ must already belong to the same precedence group. Otherwise, the *bin-op*$_i$ form a new precedence group. The associativity of the *bin-op*$_i$ is as specified by *associativity*, if present. If absent, then the associativity of the *bin-op*$_i$ is the same as the *bin-op*$_{Wi}$, if provided, and non-associative otherwise. As illustrated by the example above, the ordering of two precedence groups may be redundantly specified. Cycles in the tighter-binding-than relation on precedence groups are not allowed. All operators in the same precedence group must have the same associativity.

Taken together, precedence declarations form a partial order on groups of infix operators. Parentheses may be omitted if adjacent infix operators are ordered according to the precedence declarations, or if adjacent infix operators are from the same precedence group and the precedence group has either left- or right-associativity. Otherwise, parentheses must be included. For example, in the expression

```
v ! (i + 1) < (v ! i) + 1
```

the parentheses around `i+1` and `v!i` are required, since `!` and + are not ordered by the above precedence declarations. However, both `!` and + are more tightly binding than <, so no additional parentheses are required.

In Diesel, a declaration is visible throughout its scope, including textually earlier code within the scope. This applies to precedence declarations as well, somewhat complicating parsing. The implementation strategy used in the UW Diesel system parses expressions involving binary operators into a list of operators and operands, and these lists are converted into a traditional parse tree form only after all visible declarations have been processed.

Precedence declarations apply to infix message *names*, not to particular *functions*. All message send expressions in the scope of a given precedence declaration follow that declaration's properties.*

2.9 Method Lookup

This section details the semantics of multi-method lookup, beginning with a discussion of the motivations and assumptions that led to the semantics.

2.9.1 Philosophy

All computation in Diesel is accomplished by sending messages to objects. The lion's share of the semantics of message passing specifies method lookup, and these method lookup rules typically reduce to defining a search of the inheritance graph. In single inheritance languages, method lookup is straightforward. Some object-oriented languages, including Diesel, support multiple inheritance to allow more flexible forms of code inheritance and/or subtyping. However, multiple inheritance introduces the possibility of ambiguity during method lookup: two methods with the

* In the current UW Diesel implementation, all precedence declarations are considered to have global scope, no matter what scope they are actually written in.

same name may be inherited along different paths, thus forcing either the system or the programmer to determine which method to run or how to run the two methods in combination. Multiple dispatching introduces a similar potential ambiguity even in the absence of multiple inheritance, since two methods with differing argument specializers could both be applicable but neither be uniformly more specific than the other. Consequently, the key distinguishing characteristic of method lookup in a language with multiple inheritance and/or multiple dispatching is how exactly this ambiguity problem is resolved.

Some languages resolve all ambiguities automatically. For example, Flavors [Moon 86] linearizes the class hierarchy, producing a total ordering on classes, derived from each class' local left-to-right ordering of superclasses, that can be searched without ambiguity just as in the single inheritance case. However, linearization can produce unexpected method lookup results, especially if the program contains errors [Snyder 86]. CommonLoops [Bobrow *et al.* 86] and CLOS extend this linearization approach to multi-methods, totally ordering multi-methods by prioritizing argument position, with earlier argument positions completely dominating later argument positions. Again, this removes the possibility of run-time ambiguities, at the cost of automatically resolving ambiguities that may be the result of programming errors.

Diesel takes a different view on ambiguity, motivated by several assumptions:

- We expect programmers will sometimes make mistakes during program development. The language should help identify these mistakes rather than mask or misinterpret them.
- Our experience leads us to believe that programming errors that are hidden by such automatic language mechanisms are some of the most difficult and time-consuming to find.
- Our experience also encourages us to strive for the simplest possible inheritance rules that are adequate. Even apparently straightforward extensions can have subtle interactions that make the extensions difficult to understand and use [Chambers *et al.* 91].
- Complex inheritance patterns can hinder future program evolution, since method lookup can depend on program details such as parent ordering and argument ordering, and it usually is unclear from the program text which details are important for a particular application.

Accordingly, we have striven for a very simple system of multiple inheritance and multiple dispatching for Diesel.

2.9.2 Semantics

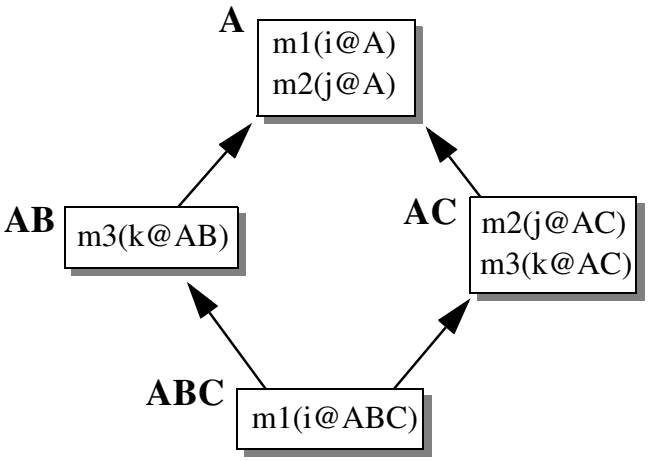
Method lookup in Diesel uses a form of Touretzky's inferential distance heuristic [Touretzky 86], summarized as "children override parents." The method lookup rules interpret a program's inheritance graph as a partial ordering on objects and classes, where being lesser in the partial order corresponds to being more specific: an object or class *A* is less than (more specific than) another object or class *B* in the partial order if and only if *A* is a proper descendant of *B*. This ordering on objects and classes in turn induces an analogous ordering on a function's set of methods specialized on the objects and classes, reflecting which of a function's methods override which other methods. In the partial ordering on methods in a function (including the implicit unspecialized method if the function's declaration includes a body), one method *M* is less than (more specific than) another method *N* if and only if each of the argument specializers of *M* is equal to or less than (more specific

than) the corresponding argument specializer of *N* (a specialized argument is strictly more specific than an unspecialized argument). Since two methods cannot have the same argument specializers, at least one argument specializer of *M* must be strictly less than (more specific than) the corresponding specializer of *N*. The ordering on methods is only partial since ambiguities are possible.

Given the partial ordering on methods, method lookup is straightforward. For a particular message send, the system locates the lexically nearest function declaration with the same name and number of arguments as the message. The system then computes the partial ordering of methods in this function. The system then throws out of the ordering any method that has an argument specializer that is not equal to or an ancestor of the corresponding actual argument object passed in the message; such a method is not applicable to the actual call. Finally, the system attempts to locate the single most-specific method remaining, i.e., the sole method that is least in the partial order over applicable methods. If no methods are applicable, then the system reports a "message not understood" error. If more than one method is applicable, but there is no single method that is more specific than all other applicable methods, then the system reports a "message ambiguous" error. Otherwise, there is exactly one method that is applicable and strictly more specific than all other applicable methods, and this method is returned as the result of the message lookup.

2.9.3 Examples

For example, consider the following inheritance graph (containing only singly-dispatched methods for the moment):



The partial ordering on classes in this graph defines `ABC` to be more specific than either `AB` or `AC`, and both `AB` and `AC` are more specific than `A`. Thus, methods defined for `ABC` will be more specific (will override) methods defined in `A`, `AB`, and `AC`, and methods defined in either `AB` or `AC` will be more specific (will override) methods defined in `A`. The `AB` and `AC` classes are mutually unordered, and so any methods defined for both `AB` and `AC` will be unordered.

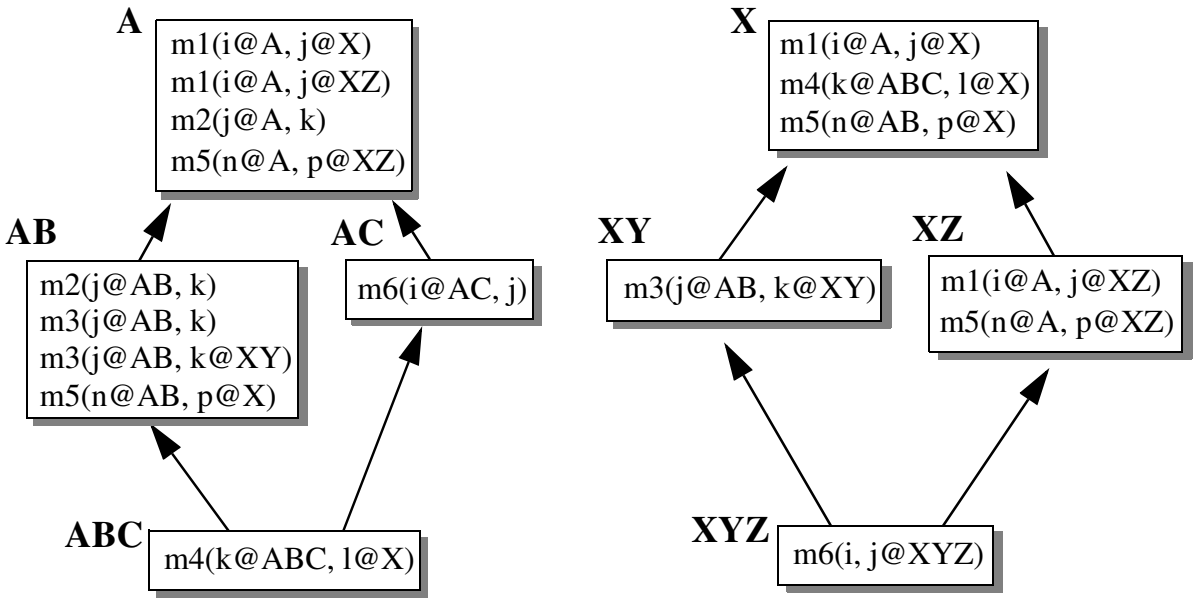
If the message `m1` is sent to the `ABC` class, both the implementation of `m1` whose formal argument is specialized on the `ABC` class and the implementation of `m1` specialized on `A` will apply, but the method specialized on `ABC` will be more specific than the one specialized on `A` (since `ABC` is more

specific than A), and so ABC's m1 will be chosen. If instead the m1 message were sent to the AB class, then the version of m1 specialized on the A class would be chosen; the version of m1 specialized on ABC would be too specific and so would not apply.

If the m2 message is sent to ABC, then both the version of m2 whose formal argument is specialized on A and the one whose formal is specialized on AC apply. But the partial ordering places the AC class ahead of the A class, and so AC's version of m2 is selected.

If the m3 message is sent to ABC, then both AB's and AC's versions of m3 apply. Neither AB nor AC is the single most-specific class, however; the two classes are mutually incomparable. Since the system cannot select an implementation of m3 automatically without having a good chance of being wrong and so introducing a subtle bug, the system therefore reports an ambiguous message error. The programmer then is responsible for resolving the ambiguity explicitly, typically by writing an overriding method in the ABC subclass which resends the message to a particular ancestor; resends are described in section 2.10. Sends of m3 to either AB or AC would be unambiguous, since the other method would not apply.

To illustrate these rules in the presence of multi-methods, consider the following inheritance graph (methods dispatched on two arguments are shown twice in this picture):



Methods m1 in A and m3 in AB illustrate that multiple methods with the same name and number of arguments may be associated with (specialized on) the same class, as long as some other arguments are specialized differently. The following table reports the results of several message sends using this inheritance graph.

message	invoked method or error	explanation
m1(ABC, XYZ)	m1(i@A, j@XZ)	XZ overrides X
m2(ABC, XYZ)	m2(j@AB, k)	AB overrides A

message	invoked method or error	explanation
m3(ABC, XYZ)	m3(j@AB, k@XY)	XY overrides unspecialized
m4(AB, XY)	"message not understood"	ABC too specific for AB ⇒ no applicable method
m5(ABC, XYZ)	"message ambiguous"	AB overrides A but XZ overrides X ⇒ no single most-specific applicable method
m6(ABC, XYZ)	"message ambiguous"	AC overrides unspecialized but XYZ overrides unspecialized ⇒ no single most-specific method

2.9.4 Strengths and Limitations

The partial ordering view of multiple inheritance has several desirable properties:

- It is simple. It implements the intuitive rule that children override their parents (they are lesser in the partial ordering), but does not otherwise order parents or count inheritance links or invoke other sorts of complicated rules.
- Ambiguities are not masked. These ambiguities are reported back to the programmer at message lookup time before the error can get hidden. If the programmer has included static type declarations, the system will report the ambiguity at type-check-time.
- This form of multiple inheritance is robust under programming changes. Programmers can change programs fairly easily, and the system will report any ambiguities which may arise because of programming errors. More complex inheritance rules tend to be more brittle, possibly hindering changes to programs that exploit the intricacies of the inheritance rules and hiding ambiguities that reflect programming errors.
- Diesel's partial ordering view of multiple inheritance does not transform the inheritance graph prior to determining method lookup, as does linearization. This allows programmers to reason about method lookup using the same inheritance graph that they use to write their programs.

Of course, there may be times when having a priority ordering over parents or over argument positions would resolve an ambiguity automatically with no fuss. For these situations, it might be nice to be able to inform the system about such preferences. An early version of Self included a prioritized multiple inheritance strategy that blended ordered and unordered inheritance, but it had some undesirable properties (such as sometimes preferring a method in an ancestor to one in a child) and interacted poorly with resends and dynamic inheritance. More recent versions of Self have greatly simplified multiple inheritance semantics, dropping prioritized inheritance. These semantics are similar to Diesel's, except that Self omits the "children-override-parents" global rule. This has the effect of declaring as ambiguous messages such as m2(ABC) in the first example in section 2.9.3. It may be that Diesel could support something akin to prioritized multiple inheritance (and perhaps even a prioritized argument list), but use these preferences as a last resort to resolving ambiguities; only if ambiguities remain after favoring children over parents would preferences on parents or argument position be considered. Such as design appears to have fewer drawbacks than the early Self approach or CLOS's approach while gaining most of the benefits.

An alternative approach might be to support explicit declarations that one method is intended to override another method. These declarations would add relations to the partial order over methods, potentially resolving ambiguities. This approach has the advantage that it operates directly on the method overriding relationship rather than on parent order or the like which only indirectly affects method overriding relationships. Moreover, this approach can only resolve existing ambiguities, not change any existing overriding relationships, thereby making it easier to reason about the results of method lookup. To implement this approach, a mechanism for naming particular methods (e.g., the method's name and its specializers) must be added.

discuss traits, Java's MI, and other more modern things

2.9.5 Multiple Inheritance of Fields

In other languages with multiple inheritance, in addition to the possibility of name clashes for methods, the possibility exists for name clashes for instance variables. Some languages maintain separate copies of instance variables inherited from different classes, while other languages merge like-named instance variables together in the subclass. The situation is simpler in Diesel, since all access to instance variables is through field accessor methods. Each field and field method declaration introduces its own internal storage table, separate from all other fields', and so distinct fields with the same name are not merged automatically. Accesses to these fields are mediated by their accessor methods, and the normal multiple inheritance rules are used to resolve any ambiguities among like-named field accessor methods.

2.9.6 Cyclic Inheritance

In the current version of Diesel, inheritance is required to be acyclic. However, cycles in the inheritance graph would be easy to allow. Instead of defining a partial order over classes, inheritance would define a preorder, where all classes participating in a cycle are considered to inherit from all other classes in the cycle, but not be strictly more specific than any of them. This preorder on classes would induce a corresponding preorder on methods. The same rules for successful method lookup still apply: a single most-specific applicable method must be found. If two methods are in a cycle in the method specificity preorder, then neither is more specific than the other. In effect, classes can participate in inheritance cycles if they define disjoint sets of methods. This design of "mutually-recursive" classes could be used to factor a single large class into multiple separate classes, each implementing a separate facet of the original class's implementation.

2.9.7 Method Invocation

If method lookup is successful in locating a single target method without error, the method is invoked. A new activation record is created, formals in the new scope are initialized with actuals, the statements within the body of the method are executed in the context of this new activation record (or the primitive method is executed, or the field accessor method is executed), the result of the method (possibly `void`) is computed, and the result is either returned normally back to the called or returned non-locally to the caller of the (`eval`) method's lexically enclosing non-nested method.

2.10 Resends

Most object-oriented languages allow one method to override another method while preserving the ability of the overriding method to invoke the overridden version: Smalltalk-80 and Java have `super`, C# has `base`, CLOS has `call-next-method`, C++ has qualified messages using the `::` operator, Trellis has qualified messages using the `'` operator, and Self has undirected and directed `resend` (integrating unqualified `super`-like messages and qualified messages). Such a facility allows a method to be defined as an incremental extension of an existing method by overriding it with a new definition and invoking the overridden method as part of the implementation of the overriding method. This same facility also allows ambiguities in message lookup to be resolved by explicitly forwarding the message to a particular ancestor.

Diesel includes a construct for resending messages that adapts the Self undirected and directed resend model to the multiply-dispatched case. The syntax for a resend is as follows:

```
resend          ::= "resend" [ "(" resend_args ")" ]
resend_args     ::= resend_arg { "," resend_arg }
resend_arg      ::= expr                        corresponding formal of sender must be
                                                  unspecialized
                  | name                        undirected resend
                  | name "@" class_ref          directed resend
```

The purpose of the resend construct is to allow a method to invoke one of the methods (including the implicit method derived from a `fun` declaration with a body) that the resending method overrides. Consequently, only methods in the same function (which have the same name and number of arguments as the resending method) whose argument specializers are ancestors of the resending method's argument specializers are considered possible targets of a resend; there is no facility for invoking a method in a different function other than regular dynamically dispatched message sending.

To invoke an overridden method, the normal prefix message sending syntax is used but with the following changes and restrictions:

- Syntactically, the name of the message is the keyword `resend`; semantically, the name of the message is implicitly the same as the name of the resending method.
- The number of arguments to the message must be the same as for the resending method.
- All specialized formal arguments of the resending method must be passed through unchanged as the corresponding arguments to the resend.

As a syntactic convenience, if all formals of the sender are passed through as arguments to the resend unchanged, then the simple `resend` keyword without an argument list is sufficient.

The semantics of a resent message are similar to a normal message, except that only methods that are greater than (less specific than) the resending method in the partial order over methods are considered possible matches; this has the effect of "searching upwards" in the inheritance graph to find the single most-specific method that the resending method overrides. The restrictions on the name, on the number of arguments, and on passing specialized objects through unchanged ensure that the methods considered as candidates are all applicable to the name and arguments of the

resend. Single-dispatching languages often have similar restrictions: Smalltalk-80 requires that the implicit `self` argument be passed through unchanged with the `super` send, and CLOS's `call-next-method` uses the same name and arguments as the calling method.

For example, the following illustrates how a resend may be used to provide an incremental extension to an existing method:

```
class ColoredRectangle isa Rectangle;

   field color(:ColoredRectangle);

   method draw(r@ColoredRectangle, d:Display):void {
      d.color := r.color; -- set the right color for this rectangle
      resend; -- do the normal rectangle drawing; sugar for resend(r, d)
   }
```

Resends may also be used to explicitly resolve ambiguities in method lookup by filtering out undesired methods. Any of the required arguments to a resend (those that are specialized formals of the resending method) may be suffixed with the `@` symbol and the name of an ancestor of the corresponding formal's specializer. This further restricts methods considered in the resulting partial order to be those whose corresponding argument specializers (if present) are equal to or ancestors of the class named as part of the resend.

To illustrate, the following method resolves the ambiguity of `area` for `Square` in favor of the `Rectangle` version:

```
abstract class Shape;

   fun area(:Shape):num;

class Rectangle isa Shape;

   method area(r@Rectangle):num { r.width * r.height }

class Rhombus isa Shape;

   method area(r@Rhombus):num { ... }

class Square isa Rectangle, Rhombus;

   method area(s@Square):num { resend(s@Rectangle) }
```

This model of undirected and directed resends is a simplification of the Self rules, extended to the multiple dispatching case. Self's rules additionally support prioritized multiple inheritance and dynamic inheritance, neither of which is present in Diesel. Self also allows the name and number of arguments to be changed as part of the resend, and Java and C# support similar abilities. In some cases, it appears to be useful to be able to change the name of the message as part of the resend. This is an area of future exploration for Diesel.

As demonstrated by Self, supporting both undirected and directed resends is preferable to just supporting directed resends as do C++ and Trellis, since the resending code does not need to be changed if the local inheritance graph is adjusted. Since CLOS does not admit the possibility of ambiguity, it need only support undirected resends (i.e., `call-next-method`); there is no need for directed resends.

2.11 Predicate Classes

To enable inheritance and classes to be used to capture run-time varying object behavior, Diesel supports *predicate classes* [Chambers 93b]. The syntax for a predicate class declaration is as follows:

```
predicate_decl ::= "predicate" name ["isa" class_refs] ["when" expr]
                       {pragma} ";"
```

(Additional declarations related to type-checking in the presence of predicate classes are described in section 3.4.4.)

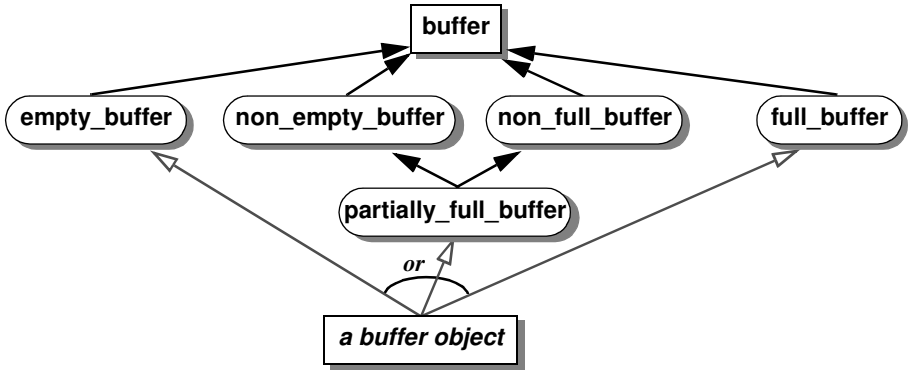
Predicate class declarations are similar to normal class declarations except that they optionally can specify an associated predicate expression. Like regular classes, methods can specialize on predicate classes. However, a predicate class is never instantiated *explicitly*. Instead, the system *implicitly* treats an object as being an instance of (i.e., inherits from) a predicate class whenever the object is already an instance of the predicate class's superclasses and (if present) evaluating the predicate expression on the object returns true.

Predicate classes allow a form of state-based dynamic classification of objects, enabling better factoring of code using declarative methods in place of imperative nested conditionals. Also, predicate classes and multi-methods allow a pattern-matching style to be used to implement cooperating methods.

For example, predicate classes could be used to implement a bounded buffer of integers:

```
class buffer isa collection[int];
   field elements(b:buffer):queue[int] { new_queue[int]() }
   field max_size(b:buffer):int;                        -- the bound on the buffer size
   fun length(b:buffer):int { b.elements.length }       -- number of current elements
   fun is_empty(b:buffer):bool { b.length = 0 }
   fun is_full(b:buffer):bool { b.length = b.max_size }
   fun get(b:buffer):int;              -- implemented below
   fun put(b:buffer, v:int):void;      -- implemented below

predicate empty_buffer isa buffer when buffer.is_empty;
   method get(b@empty_buffer):int { ... } -- raise error or block caller

predicate non_empty_buffer isa buffer when not(buffer.is_empty);
   method get(b@non_empty_buffer):int { dequeue(b.elements) }

predicate full_buffer isa buffer when buffer.is_full;
   method put(b@full_buffer, x:int):void { ... } -- raise error or block caller

predicate non_full_buffer isa buffer when not(buffer.is_full);
   method put(b@non_full_buffer, x:int):void { enqueue(b.elements, x); }

predicate partially_full_buffer isa non_empty_buffer, non_full_buffer;
```

The following diagram illustrates the inheritance hierarchy created by this example (the explicit inheritance link from the buffer object to `buffer` is omitted):



Predicate classes increase expressiveness for this example in two ways. First, important states of bounded buffers, e.g., empty and full states, are explicitly identified in the program and named. Besides documenting the important conditions of a bounded buffer, the predicate classes remind the programmer of the special situations that code must handle. This can be particularly useful during maintenance phases as code is later extended with new functionality. Second, attaching methods directly to states supports better factoring of code and eliminates `if` and `case` statements, much as does distributing methods among classes in a traditional object-oriented language. In the absence of predicate classes, a method whose behavior depended on the state of an argument object would include an `if` or `case` statement to identify and branch to the appropriate case; predicate classes eliminate the clutter of these tests and clearly separate the code for each case. In a more complete example, several methods might be associated with each special state of the buffer. By factoring the code, separating out all the code associated with a particular state or behavior mode, we hope to improve the readability and maintainability of the code. Thus, predicate classes provide a good way to implement the Strategy design pattern [**ref**].

2.11.1 Predicate Classes and Inheritance

For normal objects, an object is an instance of (i.e., inherits from) a class exactly when the relationship is defined explicitly by the programmer, and the relationship never changes at run-time. Predicate classes, on the other hand, support a form of automatic property-based classification: an object *O* is automatically considered to inherit from a predicate class *P* exactly when the following two conditions are satisfied:

- the object *O* inherits from each of the superclasses of the predicate class *P*, and
- the predicate expression of the predicate class *P* evaluates to true, when evaluated in a scope where each of the names of the predicate class's superclasses is bound to the object *O*.

By evaluating the predicate expression in a context where the superclass names refer to the object being tested, the predicate expression can query the value or state of the object.

Since the state of an object can change over time (e.g., an object's fields can be mutable), the results of predicate expressions evaluated on the object can change. If this happens, the system will automatically reclassify the object, recomputing its implicit inheritance links. For example, when

a buffer object becomes full, the predicates associated with the `non_full_buffer` and `full_buffer` predicate classes both change, and the inheritance graph of the buffer object is updated. As a result, different methods may be used to respond to messages, such as the `put` message in the filled buffer example. Predicate expressions are evaluated lazily as part of method lookup, rather than eagerly as the state of an object changes. Only when the value of some predicate expression is needed to determine the outcome of method lookup is the predicate evaluated. A separate paper describes efficient implementation schemes for predicate classes [Chambers 93b].

Because whether an object is classified as being an instance of a predicate class can change over time, predicate classes cannot be used as static types. This precludes a function being declared that specifies a predicate class as an argument type. A method in a function can specialize on a predicate class, but the function itself must be declared to accept a non-predicate superclass of the predicate class, and the typechecker will require that the function be implemented for all possible run-time states of instances of the non-predicate superclass.

A predicate class can inherit from another predicate class, thus acting as a special case of the predicate superclass. This is because an object will only be classified as an instance of the predicate subclass when it already has been classified as an instance of the predicate superclass. In essence, the superclass's predicate expression is implicitly conjoined with the subclass's predicate expression. A non-predicate class also may inherit explicitly from a predicate class, with the implication that the predicate expression will always evaluate to true for the child object; the system verifies this assertion dynamically. For example, an unbounded buffer object might inherit explicitly from the `non_full_buffer` predicate class.

A predicate class need not have a `when` clause, as illustrated by the `partially_full_buffer` predicate class defined above. Such a predicate class may still depend on a run-time condition if at least one of its superclasses is a predicate class. In the above example, the `partially_full_buffer` predicate class has no explicit predicate expression, yet since an object only inherits from `partially_full_buffer` whenever it already inherits from both `non_empty_buffer` and `non_full_buffer`, the `partially_full_buffer` predicate class effectively repeats the conjunction of the predicate expressions of its parents, in this case that the buffer be neither empty nor full.

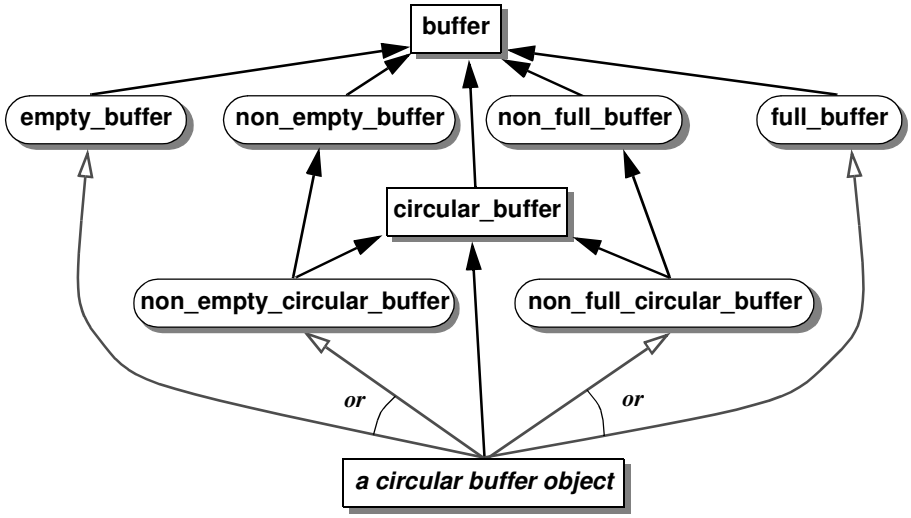
Predicate classes are intended to interact well with normal inheritance among data abstractions. If an abstraction is implemented by inheriting from some other implementation, any predicate classes that specialize the parent implementation will automatically specialize the child implementation whenever it is in the appropriate state. For example, a new implementation of bounded buffers could be built that used a fixed-length array with insert and remove positions that cycle around the array:*

```
class circular_buffer isa buffer;
   field array(b:circular_buffer):vector[int] {     -- a fixed-length array of elements
      new_vector[int](b.max_size) }
```

* This implementation ignores the buffer's `elements` field. In practice a more efficient implementation would break up `buffer` into an abstract superclass and two concrete subclasses, one for the queue-based implementation and one for the circular array implementation.

```
   var field insert_pos(b:circular_buffer):int { 0 } -- an index into the array
   var field remove_pos(b:circular_buffer):int { 0 } -- another index
   method length(b@circular_buffer) {
      (b.insert_pos - b.remove_pos) % b.array.length }

predicate non_empty_circular_buffer isa circular_buffer, non_empty_buffer;
   method get(b@non_empty_circular_buffer):int {
      let x := fetch(b.array, b.remove_pos);
      b.remove_pos := (b.remove_pos + 1) % b.array.length;
      x }

predicate non_full_circular_buffer isa circular_buffer, non_full_buffer;
   method put(b@non_full_circular_buffer, x:int):void {
      store(b.array, b.insert_pos, x);
      b.insert_pos := (b.insert_pos + 1) % b.array.length; }
```

The following diagram illustrates the extended inheritance graph for bounded and circular buffers (the `partially_full_buffer` predicate class is omitted):



Since the `circular_buffer` class inherits from the original `buffer` class, a `circular_buffer` instance will automatically inherit from the `empty_buffer` or `full_buffer` predicate class whenever the `circular_buffer` happens to be in one of those states. No `empty_circular_buffer` or `full_circular_buffer` classes need to be implemented if specialized behavior is not needed. The `non_empty_circular_buffer` and `non_full_circular_buffer` predicate classes are needed to override the default `get` and `put` methods in the non-blocking states. Any object that inherits from `circular_buffer` and that also satisfies the predicate associated with `non_empty_buffer` will automatically be classified as a `non_empty_circular_buffer`.

An interesting semantic question is which superclasses of an object should be classified as implicitly inheriting from (and therefore overriding) an otherwise unrelated predicate class. For example, in the diagram above, what inheritance relationship (if any) should exist between `circular_buffer` and `empty_buffer` with respect some some empty circular buffer object? A number of plausible alternatives exist. One simple choice would say that there is no

relationship, and so methods in the same function defined on `circular_buffer` and `empty_buffer` would be mutually ambiguous.

A second alternative would consider `circular_buffer` to implicitly inherit from `empty_buffer`, and in general a non-predicate class would implicitly inherit from any "cousin" predicate class.* This semantics would allow a regular class to gain control over the implementation choices of its superclasses, including any of their predicate subclasses. Diesel uses this second semantics.

A third alternative would reverse this decision, and have predicate classes implicitly inherit from "cousin" regular classes. This semantics treats division of an object's implementation into predicates as a "sticky" property, preserved through inheritance. Under this semantics, the buffer code could be simplified somewhat, as follows:

```
class buffer isa collection[int];
   ... -- elements, length, etc.
   fun get(b:buffer):int { dequeue(b.elements) }
   fun put(b:buffer, v:int):void { enqueue(b.elements, x); }

predicate empty_buffer isa buffer when buffer.is_empty;
   method get(b@empty_buffer) { ... } -- raise error or block caller

predicate full_buffer isa buffer when buffer.is_full;
   method put(b@full_buffer, x) { ... } -- raise error or block caller

class circular_buffer isa buffer;
   ... -- array, insert_pos, length, etc.
   method get(b@circular_buffer):int {
      var x := fetch(b.array, b.remove_pos);
      b.remove_pos := (b.remove_pos + 1) % b.array.length;
      x }
   method put(b@circular_buffer, x:int):void {
      store(b.array, b.insert_pos, x);
      b.insert_pos := (b.insert_pos + 1) % b.array.length; }
```

The non-blocking versions of `get` and `put` could be associated with the `buffer` class directly, and the `non_empty_buffer`, `non_full_buffer`, and `partially_full_buffer` predicate subclasses could be removed (if desired). The non-blocking `get` and `put` routines for circular buffers could similarly be moved up to the `circular_buffer` class itself, with the `non_empty_circular_buffer` and `non_full_circular_buffer` predicate subclasses being removed also. If the methods attached to the `empty_buffer` object were considered to override those of the `circular_buffer` object, then sending `get` to a circular buffer that was empty would (correctly) invoke the `empty_buffer` implementation. In the current semantics of predicate classes in Diesel, however, the `circular_buffer`'s implementation of `get` overrides `empty_buffer`'s, leading to an error.

* One class is a cousin of another if they share a common superclass but are otherwise unrelated.

2.11.2 Predicate Classes and Field Methods

Just as with other methods, field methods may specialize on a predicate class. Since a field method semantically allocates an internal storage table to hold the value of the field for each object inheriting the accessor (either per-object or shared), declaring a field method on a predicate class has the effect of reserving space for the field in any object that might be classified as an instance of the predicate class. The value stored in the field persists even when the field is inaccessible. When an object is created, an initial value may be provided for any fields potentially inherited from a predicate superclass, even if those fields may not be visible in the newly-created object. The semantics of accessing a field attached to a predicate class is governed by the semantics of accessing its corresponding accessor methods.

The following example exploits this semantics to implement a graphical window object that can be either expanded or iconified. Each of the two important states of the window remembers its own independent screen location and visual image (using distinct field methods in the `position` and `image` functions), and this data persists across openings and closings of the window:

```
class Window;
   var field iconified(:Window):bool { false }
   fun position(:Window):Point;
   fun set_position(:Window, :Point):void;
   fun image(:Window):Image;
   fun display(w:Window):void {
      Screen.draw(w.image, w.position); }
   fun erase(w:Window):void {
      Screen.clear(size(w.image), w.position); }
   fun move(w:Window, new_position:Point):void {
      w.erase; w.position := new_position; w.display; }
   fun iconify(w:Window):void {
      w.erase; w.iconified := true; w.display; }
   fun expand(w:Window):void {
      w.erase; w.iconified := false; w.display; }

predicate ExpandedWindow isa Window when not(Window.iconified);
   var field method position(@ExpandedWindow):Point;
   field method image(@ExpandedWindow):Text;   -- Text subtypes Image
   method expand(w@ExpandedWindow):void { }    -- override to be a no-op

predicate IconifiedWindow isa Window when Window.iconified;
   var field method position(@IconifiedWindow):Point;
   field method image(@IconifiedWindow):Icon;   -- Icon subtypes Image
   method iconify(w@IconifiedWindow):void { }   -- override to be a no-op

fun create_window(open_position:Point, iconified_position:Point,
                  text:Text, icon:Icon):Window {
   new Window {
      iconified := false,
      position@ExpandedWindow := open_position,
      position@IconifiedWindow := iconified_position,
```

```
    image@ExpandedWindow := text,
    image@IconifiedWindow := icon } }
```

A `Window` object has two `position` field methods, each storing a `Point` object, but only one is visible at a time. This allows the `display`, `erase`, and `move` routines to send the message `position` as part of their implementations, without needing to know whether the window is open or closed. The `create_window` method initializes both `position` fields when the window is created, even though the position of the icon is not visible initially. The `position@`*class* notation used in the field initialization resolves the ambiguity between the two `position` field methods. The `image` field methods are handled similarly.

Just as a function declaration cannot have a predicate class as an argument type, so a field declaration cannot have a predicate class as its argument type, since the field declaration implicitly introduces a function declaration. Field method declarations can legally specialize on a predicate class, however, as shown above.

2.12 Primitive Declarations

Declarations written in an external language may be included at top-level using a primitive declaration:

```
prim_decl       ::= prim_body ";"
```

This construct allows declarations from other languages to be included outside of any compiled routines. Primitive declarations can be used to include external-language global declarations, such as C++ `#include` directives, which may be used by later primitive functions in the same file. The detailed semantics of this construct are implementation-specific.

2.13 Pragmas

Pragmas can be used by the Diesel programmer to provide additional information and implementation directives to the Diesel implementation. The set of recognized pragmas and their interpretation is implementation-dependent.

Pragmas are written as follows:

```
pragma          ::= "(**" exprs "**)"
```

The body of a pragma uses the syntax of a Diesel expression, but its interpretation is different (and implementation-dependent).

Pragmas serve a similar role as annotations in C# and Java.

3 Static Types

Diesel supports a static type system which is layered on top of the dynamically-typed core language described in section 2. This section describes Diesel's static type system in the absence of parameterization; section 4 extends this section to cope with parameterized classes and functions. Section 3.1 presents the major goals for the type system. Section 3.2 presents the overall model of types and signatures, and section 3.3 describes the kinds of types that can be expressed in Diesel. Sections 3.4, 3.5, and 3.4.4 detail the type-checking rules for the language. Section 3.6 describes how the language supports mixed statically- and dynamically-typed code.

3.1 Goals

Static type systems historically have addressed many concerns, ranging from program verification to improved run-time efficiency. Often these goals conflict with other goals of the type system or of the language, such as the conflict between type systems designed to improve efficiency and type systems designed to allow full reusability of statically-typed code.

The Diesel type system is intended to provide the programmer with extra support in two areas: machine-checkable documentation and early detection of some kinds of programming errors. The first goal is addressed by allowing the programmer to annotate variable declarations, function arguments, and function results with explicit type declarations. These declarations help to document the interfaces to abstractions, and the system can ensure that the documentation does not become out-of-date with respect to the code it is documenting. (Type inference may be useful as a programming environment tool for introducing explicit type declarations into untyped programs.)

The Diesel type system also is intended to help detect programming errors at program definition time rather than later at run-time. These statically-detected errors include "message not understood" and "message ambiguous." The type system is designed to verify that there is no possibility of any of the above errors in programs, guaranteeing type safety but possibly reporting errors that are not actually a problem for any particular execution of the program. To make work on incomplete or inconsistent programs easier, type errors are considered warnings, and the programmer always is able to run a program that contains type errors. Dynamic type checking at run-time is the final arbiter of type safety.

Diesel's type system is not intended to improve run-time efficiency. For object-oriented languages, the goal of reusable code is often at odds with the goal of efficiency through static type declarations; efficiency usually is gained by expressing additional representational constraints as part of a type declaration that artificially limit the generality of the code. Diesel's type system strives for specification only of those properties of objects that affect program correctness, i.e., the interfaces to objects, and not of how those properties are implemented. To achieve run-time efficiency, Diesel relies on advanced implementation techniques [e.g., Dean & Chambers 94, Dean *et al.* 95a, Dean *et al.* 95b, Grove *et al.* 95, Grove 95, **more**].

Finally, Diesel's type system is *descriptive* rather than *prescriptive*. The semantics of a Diesel program are determined completely by the dynamically-typed core of the program. Type declarations serve only as documentation and partial redundancy checks; they do not influence the

execution behavior of programs. This is in contrast to some type systems, such as Dylan's, where an argument type declaration can mean a run-time type check in some contexts and act as a method lookup specializer in other contexts.

3.2 Types and Signatures

The design of the Diesel type system is affected strongly by certain language features. Foremost of these is the separation of classes and functions, and the support for multiple dispatching. Type systems for single dispatching languages where methods are components of classes use types that "contain" a list of legal operations. In Diesel, however, functions are defined separately from classes, and methods in those functions can specialize any subset of their arguments to any classes that descend from the function's argument types. Consequently, types in Diesel do not "contain" their legal operations, but instead these are specified separately via signatures.

A *type* in Diesel describes a set of possible objects, which are said to *conform* to the type. Each non-predicate class* declaration introduces a corresponding type, called a class type, distinct from all other types; the set of objects conforming to a class type is exactly the set of objects that are equal to or inherit directly or indirectly from the corresponding class. One type may be a *subtype* of another, meaning that all objects that conform to the subtype also conform to the supertype; the set of objects conforming to a subtype is thus a subset of the set of objects conforming to a supertype. Subtyping is reflexive and transitive; we say that a type is a *proper subtype* of another if the first is a subtype of the second, but not vice versa. The type of a class is a proper subtype of each of the types of the class's superclasses. Since inheritance is acyclic, the subtyping relation over class types forms a partial order. As described in subsection 3.3, additional sorts of types augment the class types and the subtyping partial order.

A *signature* in Diesel defines an allowed invocation interface of a function, specifying a sequence of argument types and a corresponding result type. A function can have mutliple signatures, each of which specifies a legal way the function can be invoked. A function's declaration generates its first signature, and additional signatures can be generated throigh `signature` declarations and method and field method declarations having `signature` annotations.

The interface of a type, i.e., the set of operations that can be performed on objects conforming to the type, is then the set of signatures that mention that type (or a supertype) as one of their argument types.

Types and signatures represent a contract between clients and implementors that enable function calls and function implementations to be type-checked. The presence of a signature for a function licenses clients to invoke that function with arguments that conform to the corresponding argument types in the signature, and guarantees that the result of such an invocation will conform to the result type appearing in the signature. Clients are obligated to only invoke functions in licensed ways; if no signature is present to license a function call, the type-checker will report a "message may not be understood" warning. Correspondingly, the presence of a signature obligates the set of methods

* Except where noted explicitly, in this section a named object is viewed as a special kind of concrete class.

implementing the function to validate the signature's promises to clients. To achieve this, the collection of methods implementing a signature must be *conforming*, *complete*, and *consistent*:

- Conformance implies that each method implementing a signature has unspecialized argument types that are supertypes of the corresponding argument types of the signature and a result type that is a subtype of the signature's result type. Conformance is Diesel's version of the standard contravariance rule found in singly-dispatched statically-typed languages.
- Completeness implies that the methods must handle all possible combinations of run-time arguments of a message declared legal by the signature.
- Consistency implies that the methods must not be ambiguous for any possible combination of run-time arguments of a message declared legal by the signature.

Checking these properties is the subject of section 3.4.2.

In Diesel and in most other object-oriented languages, the code inheritance graph and the subtyping graph are joined: a class is a subtype of another class if and only if it inherits from that other class. Sometimes this constraint becomes awkward [Snyder 86], for example when a class supports the interface of some other class or type, but does not wish to inherit any code. Other times, a class reusing another class's code cannot or should not be considered a subtype; covariant redefinition as commonly occurs in Eiffel programs is one example of this case [Cook 89].

To increase flexibility and expressiveness, Diesel's predecessor, Cecil [Chambers 92b, Chambers 93a], separates subtyping from code inheritance. In Cecil, types, subtyping, and conformance can be declared independently of classes and inheritance. However, since in most cases the subtyping graphs and the inheritance graphs are parallel, requiring programmers to define and maintain two separate hierarchies would become too onerous to be practical. To simplify specification and maintenance of the two graphs, in Cecil the programmer can specify both a type and a representation, and the associated subtyping, conformance, and inheritance relations, with a single declaration. Similarly, a single declaration can be used to specify both a signature and a method implementation. **explain why Diesel changed.**

In Diesel, each class declaration gives rise to a distinct named class type, and one class type is a subtype of another class type only when the first's class inherits from the other's. This is refered to as *nominal* or *by-name subtyping*. Treating a subclass as a legal subtype is validated by verifying that each signature is correctly implemented for all arguments conforming to the signature's argument types. An alternative approach, called *structural subtyping*, would use the intrinsic properties of types, e.g., the set of operations defined on them, to implicitly decide when one type could be treated as a subtype of another. Structural subtyping is more flexible than nominal subtyping, but structural subtyping between class types is difficult to define, because class types do not have intrinsic properties to compare: all their operations are specified separately through signatures. As a result, Diesel limits subtyping between class types to explicit inheritance between classes. Diesel does use structural subtyping for other kinds of types.

3.3 Type Expressions

The syntax of type expressions (excluding parameterization and module-related constructs) is as follows:

```
type           ::= lub_type
lub_type       ::= lub_type "|" glb_type
                |  glb_type
glb_type       ::= glb_type "&" simple_type
                |  simple_type
simple_type    ::= named_type
                |  closure_type
                |  "(" type ")"
```

3.3.1 Named Types

Types with names can be directly named:

```
named_type     ::= class_ref
```

The name-space for types is separate from the name-spaces for classes/objects/variables and functions.

As explained in section 3.2, each non-predicate class and named object declaration gives rise to a corresponding named class type. Each synonym declaration introduces a new type name as well.

In addition, the Diesel type system includes four special predefined types:

- The type `any` is implicitly a supertype of all types, thus defining the top of the type lattice. `any` may be used whenever code does not require any special operations of an object.
- The type `void` is used (and may only be used) as the result type of functions, methods, and closures that may return normally but without a useful result. As a special case, if the result type of a function, method, or closure is `void`, then its body's result expression is allowed to have any (legal) type, since the client will be ignoring it. The predefined object `void` has type `void`.
- The type `none` is implicitly a subtype of all other types, thus defining the bottom of the type lattice. It is the result type of a closure that terminates with a non-local return, since such a closure never returns to its caller. It also is the result type of the primitive `loop` method, which also never returns normally. Finally, `none` is an appropriate argument type for closures that will never be called.
- The type `dynamic` is used to disable static type checking. Any value can be bound to a variable of type `dynamic` (i.e., `dynamic` is a supertype of all other types), and a value of type `dynamic` can be used in any context (i.e., `dynamic` is a subtype of all other types).* Wherever type declarations are omitted, `dynamic` is implied (with the exception of closure results and constant local variable declarations, as described in section 3.5), which supports exploratory programming as described in section 3.6.

* Transitivity of subtyping would then mean that all types were subtypes of all types, by way of `dynamic`. To prevent this while allowing `dynamic` to be modeled in this simple way, subtyping is not transitive through `dynamic`.

3.3.2 Closure Types

The type of a closure is described using the following syntax:

```
closure_type   ::= "&" "(" [arg_types] ")" [type_decl]
arg_types      ::= arg_type { "," arg_type }
arg_type       ::= [[name] ":"] type
```

A closure type of the form

$\&(t_1,\ldots,t_N):t_R$

describes a closure whose `eval` method has the signature:

signature eval(:&(t_1,...,t_N):t_R, :t_1, ..., :t_N):t_R

Closure types are related by implicit structural subtyping rules that reflect standard contravariant subtyping: a closure type of the form $\&(t_1,\ldots,t_N):t_R$ is a subtype of a closure type of the form $\&(s_1,\ldots,s_N):s_R$ iff each t_i is a *super*type of the corresponding s_i and t_R is a *sub*type of s_R.

3.3.3 Least-Upper-Bound Types

The least upper bound of two types, $type_1 \mid type_2$, is a supertype of both $type_1$ and $type_2$, and a subtype of all types that are supertypes of both $type_1$ and $type_2$. Least-upper-bound types are most useful in conjunction with parameterized types, described in section 4.

3.3.4 Greatest-Lower-Bound Types

The greatest lower bound of two types, $type_1$ `&` $type_2$, is a subtype of both $type_1$ and $type_2$, and a supertype of all types that are subtypes of both $type_1$ and $type_2$.

Note that the greatest-lower-bound of two class types is different than the type of a class that inherits from the classes of the two class types. For example,

```
c1 & c2
```

is a different type than the type introduced by the declaration

```
class c3 isa c1, c2;
```

The type `c3` is a subtype of `c1 & c2` (all types that subtype both `c1` and `c2` are automatically subtypes of `c1 & c2`), but not identical to it. The reason is that the programmer might later define a `c4` class:

```
class c4 isa c1, c2;
```

The type `c4` is also a subtype of `c1 & c2`, but `c3` and `c4` are different and in fact mutually incomparable under the subtype relation. `c1 & c2` is a proper supertype of the types of all classes that inherit from both `c1` and `c2`.

The greatest-lower-bound and least-upper-bound type constructors serve to extend the subtyping partial order over the other kinds of types to a full lattice.

3.4 Type Checking Messages

This section describes Diesel's type checking rules for message sends and method declarations. Section 3.5 describes type checking for other, simpler kinds of expressions, as well as statements and declarations. Parameterized types are described in section 4.

In Diesel, all control structures, instance variable accesses, and basic operators are implemented via message passing, so messages are the primary kind of expression to type-check. For a message to be type-correct, there must be a single most-specific applicable method implementation defined for all possible argument objects that might be used as an argument to the message. However, instead of directly checking each message occurring in the program against the methods in the program, in Diesel messages are checked against the set of signatures defined for the argument types of the message, and separately each signature is checked that it is implemented conformingly, completely, and consistently by the methods in the function referenced by the signature.

Using signatures as an intermediary for type checking has three important advantages. First, the type-checking problem is simplified by dividing it into two separable pieces. Second, checking signatures enables all interfaces to be checked for conformance, completeness, and consistency independent of whether messages exist in the program to exercise all possible argument types. Finally, signatures enable the type checker to assign blame for a mismatch between implementor and client. If some message is not implemented completely, the error is either "message not understood" or "message not implemented correctly." If the signature is absent, it is the former, otherwise the latter. Signatures inform the type checker (and the programmer) of the intended interfaces of abstractions, so that the system may report more informative error messages. Of course, the "missing signature" error is sometimes the appropriate message to report, but the type checker cannot accurately distinguish this from the "message not understood" alternative.

Subsection 3.4.1 describes checking messages against signatures, and subsection 3.4.2 describes checking signatures against method implementations. **mention other subsections**

3.4.1 Checking Messages Against Signatures

Given a message of the form *name*`(`$expr_1$`,...,`$expr_N$`)`, where a function named *name* with N arguments has been declared and each $expr_i$ type-checks and has static type T_i, the type checker locates all signatures in scope associated with the named function (including the implicit signature derived from the function's declaration and any signatures generated as part of `method signature` and `field method signature` declarations) of the form *name*`(`S_1`,...,`S_N`):`S_R where each type S_i is a supertype of the corresponding T_i. If this set of *licensing signatures* is empty, the checker reports a "message may not be understood" error. Otherwise, the message send is considered type-correct.

To determine the type of the result of the message send, the type system calculates the most-specific result type of any licensing signature. This most-specific result type is computed as the greatest lower bound of the result types of all licensing signatures. In the absence of other type errors, this greatest lower bound will normally correspond to the result type of the most-specific signature.

To illustrate, consider the message `copy(r)`, where the static type of `r` is `Rectangle`. The following classes, functions, and signatures are assumed to be in scope:

```
abstract class Shape;
   class Rectangle isa Shape;
      class Square isa Rectangle;
   class Circle isa Shape;

fun copy(:Shape):Shape;
   signature copy(:Rectangle):Rectangle;
      signature copy(:Square):Square;
   signature copy(:Circle):Circle;
```

The signature `copy(:Circle):Circle` is not licensing, since `Rectangle`, the static type of `r`, is not a subtype of `Circle`. Neither is the signature `copy(:Square):Square`, since `r` is not known to be a subtype of `Square`. At run-time, `r` might turn out to conform to `Square`, but the static checker cannot assume this and so must ignore that signature. The `Shape` and `Rectangle` signatures are licensing, so the `copy` message is considered legal. The type of the result is known to be both a `Shape` and a `Rectangle`. The greatest lower bound of these two is `Rectangle`, so the result of the `copy` message is of type `Rectangle`.*

Unlike method dispatching, it is acceptable for more than one signature to license a message. Signatures are contracts that clients can assume, and if more than one signature licenses the message, then the client can assume more guarantees about the type of the result. The greatest lower bound is used to calculate the message's result type, rather than, say, the least upper bound, because each licensing signature can be assumed to be in force. At run-time, some single method will be selected, but that method will be required to honor the result type guarantees of all the licensing signatures, and so the target method implementation will return an object that conforms to the result types of all the licensing signatures, i.e., the greatest lower bound of these signatures. In common practice, some most-specific signature's result type will be the greatest lower bound, such as the `Rectangle` type selected above.

3.4.2 Checking Signatures Against Method Implementations

The type checker ensures that, for every signature in the program, all possible messages that could be declared type-safe by the signature would in fact locate a single most-specific method with appropriate argument and result type declarations. This involves locating all methods in the function named by the signature (including the methods implied by `fun` declarations with bodies and by `field` declarations), finding those that are *applicable* to the signature (i.e., all those that could be invoked by a message licensed by the signature), and ensuring that they *conformingly*, *completely*, and *consistently* implement the signature.

* Note that this example follows the pattern that the type of the result of `copy` is the same as the type of its argument. If this pattern were required to hold for all current and future implementations of `copy`, then a parameterized function, described in section 4, would be the appropriate way to define this function's type. If, however, not all implementations of `copy` need adhere to this pattern, then a collection of signatures defining the pattern by enumeration is appropriate.

More precisely:

- A method is considered *applicable* to a signature iff they are for the same function (and therefore have the same name and number of arguments) and there exists some tuple of argument objects that both inherit from the corresponding specializers of the method (where specified) and conform to the corresponding argument types of the signature.
- A method properly *conforms* to a signature iff
  - the type of each unspecialized formal is a supertype of the signature's corresponding argument type (no constraints are imposed on specialized formals), and
  - the method's result type is a subtype of the signature's result type.
- A set of methods *completely* implements a signature iff, for each possible tuple of argument objects that conform to the corresponding argument types in the signature, there exists at least one method in the set that is applicable to the argument objects, i.e., where the argument objects inherit from the method's specializers (where specified).
- A set of methods *consistently* implements a signature iff, for each possible tuple of argument objects that conform to the corresponding argument types in the signature, there exists at most one most-specific applicable method in the set.

There are an unbounded number of possible run-time argument objects. However, each such object is either a named object or a direct instance of a concrete class. All direct instances of a concrete class have identical behavior with respect to type-checking signature implementations, and so the type-checker can safely use the concrete class as a static *representative* of all of its direct instances. Thus, the set of possible argument objects is drawn from the (finite) set of named objects and concrete classes in the program being checked. Abstract classes are not included when considering possible argument objects, since they do not have direct instances at run-time; ignoring them when checking implementation of signatures allows them to be incompletely implemented. (Checking signature implementations in the face of predicate classes is discussed separately, in section 3.4.4.)

Proper conformance of a method to a signature can be checked in isolation of any other methods and signatures in the program. However, when classes and functions are declared separately, or in the presence of multi-methods, it is not possible to check individual methods in isolation for completeness and consistency, since interactions among abstract classes and functions and/or among multi-methods can introduce omissions or ambiguities not detectable when viewing only a subset of the declarations. Consequently, for each signature, the type checker (conceptually) enumerates all possible (static) argument objects that conform to the signature's argument types. (A more efficient algorithm to perform this checking is described elsewhere [Chambers & Leavens 94].) For each tuple of argument objects, the type checker simulates method lookup and checks that the simulated message would locate exactly one most-specific method. If no method is found, the type checker reports a "signature implemented incompletely" error. If multiple mutually ambiguous methods are found, the type checker reports a "signature implemented inconsistently" error. Otherwise, the single most-specific method has been found for those arguments. In this case, the type checker finally verifies that the argument objects conform to the declared argument types of the target method and that the declared result type of the method is a subtype of the signature's result type.

For example, consider type-checking the implementation of the following signature in the context of the following class and object declarations:

```
signature =(:Shape, :Shape):bool;
abstract class Shape;
class Circle isa Shape;
class Rectangle isa Shape;
abstract class Rhombus isa Shape;
class Square isa Rectangle, Rhombus;
object UnitSquare isa Square;
```

The type checker would first collect all possible static argument objects that conform to `Shape`. In this example, such objects are `Circle`, `Rectangle`, `Square`, and `UnitSquare`; the `Shape` and `Rhombus` classes are not included because they are `abstract`.

The type checker then enumerates all possible combinations of static argument objects conforming to the argument types in the signature, yielding the following possible messages:

```
=(Circle,Circle)
=(Circle,Rectangle)
=(Circle,Square)
=(Circle,UnitSquare)
=(Rectangle,Circle)
=(Rectangle,Rectangle)
...
=(UnitSquare,Square)
=(UnitSquare,UnitSquare)
```

For each message, method lookup is simulated to verify that the message invokes a unique most-specific applicable method, that the method's unspecialized formals (if any) are supertypes of the signature's corresponding argument types (`Shape`), and that the method returns a subtype of the signature's result type (`bool`).

3.4.3 Comparison with Other Type Systems

For singly-dispatched languages, most type systems apply contravariant rules to argument and result types when checking that an overriding method can safely be invoked in place of the overridden method: argument types in the overriding method must be supertypes of the corresponding argument types of the overridden method, while the result type must be a subtype. Diesel's type system does not directly compare one method against another to enforce contravariant redefinition rules, but instead compares a method against every signature to which it is applicable to enforce contravariant rules for non-specialized arguments. In Diesel terms, in a singly-dispatched language a signature is inferred from the superclass's method, and then all subclass methods (i.e., those methods that are applicable to the signature) are checked for conformance to the signature.

Specialized arguments need not obey contravariant restrictions. In fact, the type corresponding to a specialized argument for one method can be a subtype of the type of the corresponding argument for a more general method, because in order to be a more specific method, its argument specializers

must be more specific. This does not violate type safety because run-time dispatching will guarantee that the method will only be invoked for arguments that inherit from the argument specializer. Unspecialized arguments, on the other hand, cannot safely be covariantly redefined, because there is no run-time dispatching on such arguments ensuring that the method will only be invoked when the type declaration is correct.

Singly-dispatched languages make the same distinction between specialized and unspecialized arguments implicitly in the way they treat the type of the receiver. For most singly-dispatched languages, the receiver argument is omitted from the signatures being compared, leaving only unspecialized arguments and hence the contravariant redefinition rule. If the receiver type were included as an explicit first argument, it would have to be given special treatment and allowed to differ covariantly. In Diesel, any subset of the arguments of a method can be specialized, and different methods applicable to the same signature can specialize on different arguments, leading to type-checking rules that explicitly account for specialized vs. unspecialized arguments. If all methods in a Diesel program specialize on their first argument only, Diesel's type checking rules would reduce to those found in a traditional singly-dispatched language.

Few multiply-dispatched languages support static type systems. Two that are most relevant are Polyglot [Agrawal *et al.* 91] and Kea [Mugridge *et al.* 91]. In both of these systems, type checking of method's consistency and completeness requires that all "related" methods (all methods in the same generic function in Polyglot and all variants of a function in Kea) be available to the type checker, just as does Diesel. Neither Polyglot nor Kea supports abstract classes and abstract functions.

3.4.4 Type Checking Predicate Classes

Predicate classes are intended to represent alternative ways of implementing an object's interface. Accordingly, it should be possible to type-check programs using predicate classes, under the assumption that the particular state of the object does not affect its external interface. In particular, to guarantee type safety in the presence of predicate classes, the type checker must verify that for each message declared in the interface of some object *O*:

- at all times there is an implementation of the message inherited by the object *O*; and
- at no time are there several mutually ambiguous implementations of the message inherited by the object *O*.

These two tests correspond to extending the tests of completeness and consistency of method implementations to cope with the presence of predicate classes.

The set of methods inherited by the object *O* from normal objects is fixed at program-definition time and can be type-checked in the standard way. Methods inherited from predicate classes pose more of a problem. If two predicate classes might be inherited simultaneously by an object, either one predicate class must be known to override the other, or they must provide implementations of disjoint functions. For example, in the bounded buffer implementation described in section 2.11, since an object can inherit from both the `non_empty_buffer` and the `non_full_buffer` predicate classes, the two predicate classes should not implement methods in the same function. Similarly, if the only implementations of some message are in some set of predicate classes, then

one of the predicate classes must always be inherited for the message to be guaranteed to be understood. In other words, the checker needs to know when one predicate class *implies* another, when two predicate classes are *mutually exclusive*, and when a group of predicate classes is *exhaustive*. Once these relationships among predicate classes are determined, the rest of type-checking becomes straightforward.

Ideally, the system would be able to determine all these relationships automatically by examining the `when` expressions attached to the various predicate classes. However, `when` expressions in Diesel can run arbitrary user-defined code, and consequently the system would have a hard time automatically inferring implication, mutual exclusion, and exhaustiveness. Consequently, we rely on explicit user declarations to determine the relationships among predicate classes; the system can verify dynamically that these declarations are correct.

A declaration already exists to describe when one predicate class implies another: the `isa` declaration. If one predicate class explicitly inherits from another, then the first predicate class's `when` expression, after extending it by conjunction with all its ancestor's `when` expressions, automatically implies the second's. Any methods in the child predicate class override those in the ancestor, resolving any ambiguities between them.

Mutual exclusion and exhaustiveness are specified using declarations of the following form:

```
disjoint_decl  ::= "disjoint" names ";"
cover_decl     ::= "cover" name "by" names ";"
divide_decl    ::= "divide" name "into" names ";"
names          ::= name { "," name }
```

The disjoint declaration

disjoint $class_1$, ..., $class_n$;

signals to the static type checker that the predicate classes named by each of the $class_i$ will never be inherited simultaneously, i.e., that at most one of their predicate expressions will evaluate to true at any given time for any given object. Mutual exclusion of two predicate classes implies that the type checker should not be concerned if both predicate classes define methods with the same name, since they cannot both be inherited by an object. To illustrate, the following declarations extend the bounded buffer example of section 2.11 with mutual exclusion information:

```
disjoint empty_buffer, non_empty_buffer;
disjoint full_buffer, non_full_buffer;
```

The system can infer that `empty_buffer` and `full_buffer` are mutually exclusive with `partially_full_buffer`. Note that `empty_buffer` and `full_buffer` are not necessarily mutually exclusive.

The cover declaration

cover *class* **by** $class_1$, ..., $class_n$;

implies that whenever an object *O* descends from *class*, the object *O* will also descend from at least one of the $class_i$ predicate classes; each of the $class_i$ are expected to descend from *class* already. Exhaustiveness implies that if all of the $class_i$ implement some message, then any object inheriting

from *class* will understand the message. For example, the following coverage declaration extends the bounded buffer predicate classes:

```
cover buffer by empty_buffer, partially_full_buffer, full_buffer;
```

Often a group of predicate classes divide an abstraction into a set of exhaustive, mutually-exclusive subcases. The divide syntactic sugar makes specifying such situations easier. A declaration of the form

divide *class* **into** $class_1$, ..., $class_n$;

is syntactic sugar for the following two declarations:

disjoint $class_1$, ..., $class_n$;
cover *class* **by** $class_1$, ..., $class_n$;

(FIGURE OUT AND) EXPLAIN HOW THE EARLIER ITC ALGORITHM SHOULD BE EXTENDED

(Note: the current UW Diesel implementation does not currently implement full type-checking of predicate classes and their methods. In particular, neither completeness nor consistency is checked for any signature with an applicable method specializing on a predicate class. This allows programs involving predicate classes to be written and run, but might not catch all implementation errors statically.)

3.5 Type Checking Expressions, Statements, and Declarations

Type checking an expression determines whether it is type-correct, and if type-correct also determines the type of its result. Type checking a statement or declaration simply checks for type correctness. All constructs are type-checked in a typing context containing the following information:

- a binding from the name of each variable, class, or named object in scope to either:
  - if a variable, then the variable's declared or inferred type and an indication of whether the variable binding is assignable or constant, or
  - if a class or object, then the kind of the class or object (`abstract class`, `class`, `object`, or `predicate class` optionally with a given `when` *expr* clause);
- the set of inheritance relations between classes currently in scope;
- a binding for each type name in scope to the corresponding type;
- the set of declared direct subtyping relations between class types currently in scope;
- a binding for each function in scope, recording its name and number of arguments;
- the set of signatures currently in scope, each recording the signature's function, argument types, and result type; and
- the set of method declarations currently in scope (for type checking resends and field initializations).

In the following, we assume that desugarings involving methods have been applied, so that a `fun` declaration with a body is desugared into a `fun` declaration without a body plus a separate unspecialized `method` declaration, a `field` declaration is desugared into one or two `fun`

declarations and one or two `method` declarations whose bodies are special field accessor primitives (which cannot be expressed directly in Diesel source), and a `field method` declaration is desugared into one or two `method` declarations whose bodies are special field accessor primitives.

The type checking rules for expressions are as follows:

- A literal constant is always type-correct. The type of the result of a literal constant is the corresponding predefined class type.
- A reference *name* is type-correct iff *name* is defined in the typing context (i.e., if there exists a declaration of that name earlier in the same scope or in a lexically-enclosing scope) as either a variable or a named object. If a variable, then the type of the result is the associated type of the variable in the typing context, otherwise it is the class type corresponding to the named object.
- An object constructor expression of the general form

  `new` *class* `{` $field_1$`@`c_1 `:=` $expr_1$`,` ... `,` $field_N$`@`c_N `:=` $expr_N$ `}`

  where any of the `@`c_i may be omitted, is type-correct iff:
  - *class* names a non-abstract, non-predicate, non-`void` class or named object;
  - if `@`c_i is present, then c_i names an ancestor of *class* (if absent, it is considered to be the same as *class*);
  - each $field_i$ names a field method F_i specialized on or inherited unambiguously by c_i, ignoring any overriding non-field methods, and F_i is not `shared`;
  - each $expr_i$ is type-correct, returning an object of static type T_i, and T_i is a subtype of the type of the contents of the field F_i; and
  - no field F_i is initialized more than once.

  The type of the result of an object constructor expression is the class type corresponding to *class*.
- A closure constructor expression of the general form

  `&` `(`x_1`:`$type_1$`,` ... `,` x_N`:`$type_N$`)` `:`$type_R$ `{` *body* `}`

  is type-correct iff:
  - the x_i, where provided, are distinct;
  - each of the $type_i$, if provided, notates a non-`void` type in the current typing context, and otherwise is assumed to be `dynamic`;
  - *body* is type-correct, checked in a typing context constructed by extending the current typing context with constant variable bindings for each of the x_i to the corresponding type $type_i$; and
  - if `:`$type_R$ is omitted, then $type_R$ is inferred to be the type of the result of *body*; otherwise $type_R$ notates a type in the current typing context; and if $type_R$ is non-`void`, the type of the result of *body* is a subtype of $type_R$.

  The type of the result of a closure constructor expression of the above form is

  `&` `(`$type_1$`,` ... `,` $type_N$`)` `:`$type_R$`.`
- A vector constructor expression of the general form `[`*elemtype* $expr_1$`,`...`,`$expr_N$`]` is type-correct iff:
  - each of the $expr_i$ is type-correct, with static type T_i; and

- if *elemtype* is `:`*type*`:`, then *type* notates a non-`void` type in the current typing context and each of the T_i is a subtype of *type*; otherwise, *type* is inferred to be the least upper bound of the T_i.

The type of the result of a vector constructor expression is the predefined parameterized type `prim_i_vector[`*type*`]`. (See section 4 for information on parameterized types.)

- A message expression of the general form *name*(*expr*$_1$`, ...,` *expr*$_N$) is type-correct iff:
  - *name* names a function of *N* arguments in the current typing context;
  - each of the *expr*$_i$ is type-correct, with static type T_i; and
  - the set $S = \{S_1, ..., S_M\}$ of licensing signatures is non-empty, where *S* is the set of signatures in the current typing context of the form S_i = **`signature`** *name* (*t*$_{i1}$`, ...,` *t*$_{iN}$) `:`*t*$_{iR}$ where each T_i is a subtype of t_i.

  The type of the result of a message is the greatest lower bound of all the result types t_{iR} of the licensing signatures. (This typechecking of message expressions is also discussed in section 3.4.1. Verifying correctness of the implementation of signatures is described in subsection 3.4.2.)

- A resend expression of the general form

  `resend``(...,` *x*$_i$`@`*c*$_i$`, ...,` *expr*$_j$`, ...)`

  is type-correct iff:
  - each of the arguments x_i or *expr*$_i$ is type-correct, with static type T_i;
  - the resend is nested textually in the body of a method *M*;
  - *M* takes the same number of arguments, *N*, as does the resend;
  - for each specialized formal parameter *formal*$_i$ of *M*, specialized on *class*$_i$, the ith argument to the resend is *formal*$_i$, possibly suffixed with `@`c_i, and *formal*$_i$ is not shadowed with a local variable of the same name;
  - for each unspecialized formal parameter *formal*$_j$ of *M*, the jth argument to the resend is not suffixed with `@`c_j;
  - for each resend argument of the form *formal*$_i$`@`c_i, c_i is a proper ancestor of *class*$_i$, the specializer of *formal*$_i$, and c_i is not `void`;
  - when method lookup is simulated with a message name the same as *M* and with *N* arguments, where argument *i* is either `any` (if *formal*$_i$ of *M* is unspecialized), c_i (if the argument of the resend is directed using the `@`c_i suffix notation), or *class*$_i$, the specializer of *formal*$_i$ (otherwise), and where the resending method *M* is removed from the set of applicable methods in the current typing context, exactly one most-specific target method *R* is located; and
  - each T_i is a subtype of the type of *R*'s corresponding formal.

  The type of the result of a resend expression is the declared result type of the target method *R*.

- A parenthetical expression of the form `(` *body* `)` is type-correct iff *body* is type-correct. The type of the result of a parenthetical expression is the type of the result of *body*.

The following rules define type-correctness of statements:

- An assignment statement of the form *name* `:=` *expr*`;` is type-correct iff:
  - *expr* is type-correct, with static type T_{expr};
  - *name* is bound to an assignable variable of type T_{name} in the current typing context; and

    - T_{expr} is a subtype of T_{name}.
- An expression statement of the form *expr*`;` is type-correct iff *expr* is type-correct.

The following rules define type-correctness of return clauses:

- A normal return clause, of the form *expr*, is type-correct iff *expr* is type-correct, with static type *T*. The type of the result of the return clause is *T*.
- A non-local return clause, of the form `^` or `^` *expr* , is type-correct iff:
    - if present, *expr* is type-correct, with static type *T*; if absent, *T* is considered to be `void`;
    - the non-local return statement is nested textually inside the body of a method *M*; and
    - if the declared result type of *M* is non-`void`, then *T* is a subtype of the declared result type of *M*.

  The type of the (normal) result of a non-local return clause is `none`.

The body of a method, closure, or parenthetical expression, viewed abstractly as a possibly empty sequence of statements and declarations optionally followed by a return clause, is type-correct iff its statements, declarations, and return clause are type-correct. The extensions made to the typing context by declarations in the body are visible throughout the body. The type of the result of a body is the type of its return clause, if present, or `void`, otherwise.

The following rules define type-correctness of declarations:

- A variable declaration of the form

  `let` *var name*`:`*type* `:=` *expr*`;`

  where *var* is either `var` or empty and `:`*type* may be omitted, is type-correct iff:
    - *name* is not otherwise defined as a variable, class, or named object in the same scope;
    - *expr* is type-correct in a typing context where *name* and all variables defined later in the same scope are unbound, resulting in static type *T*;
    - if `:`*type* is omitted, then if the declaration is in a dynamic scope (inside the body of a method, closure, or parenthetical expression) and *var* is empty, then *type* is inferred to be *T*, otherwise it is inferred to be `dynamic`; if *type* is provided, then it must notate a type in the current typing context, and *T* must be a subtype of *type*.

  The typing context is extended to include a variable binding for *name* of type *type* that is assignable if *var* is `var` and constant otherwise.
- A class or named object declaration of the form

  kind name **`isa`** $superclass_1$`, ...,` $superclass_M$ `{` $field_1$`@`c_1 `:=` $expr_1$`, ...,` $field_N$`@`c_N `:=` $expr_N$ `};`

  where *kind* is `abstract class`, `class`, or `object` and where any of the `@`c_i may be omitted, is type-correct iff:
    - *name* is not otherwise defined as a variable, class, or named object in the same scope, nor is it otherwise defined as a type in the same scope;
    - *kind* is `abstract class`, `class`, or `object`;
    - each $superclass_i$ names a non-`void` class or named object;
    - no cycles are introduced into the inheritance graph;
    - if field initializers are given, then *kind* is `object`;

- if `@`c_i is present, then c_i names an ancestor of the class or object being declared (if absent, it is considered to be the same as the class or object being declared);
- each $field_i$ names a field method F_i specialized on or inherited unambiguously by c_i, ignoring any overriding non-field methods, and F_i is not `shared`;
- each $expr_i$ is type-correct, returning an object of static type T_i, and T_i is a subtype of the type of the contents of the field F_i; and
- no field F_i is initialized more than once.

The typing context is extended to include a variable binding from *name* to a new class or named object of kind *kind*, and with inheritance links from the new class or object to each of the $superclass_i$ classes. The typing context is also extended with a type binding from *name* to a new class type, and with subtyping relations from the new class type to the types corresponding to the $superclass_i$ classes (the type corresponding to a non-predicate class or named object is its corresponding class type, while the type corresponding to a predicate class is the greatest lower bound of the types corresponding to the predicate class's non-predicate ancestors).

- A class or object extension declaration of the form

  `extend` *kind name* **`isa`** $superclass_1$`, ...,` $superclass_M$
  `{` $field_1$`@`c_1 `:=` $expr_1$`, ...,` $field_N$`@`c_N `:=` $expr_N$ `};`

  where *kind* is `class` or `object` and where any of the `@`c_i may be omitted, is type-correct iff:

  - if *kind* is `class`, then *name* is bound in the typing context to a non-predicate class other than `void` and `any`;
  - if *kind* is `object`, then *name* is bound in the typing context to a named object other than `void`;
  - the same constraints on the superclass and field initialization clauses as with a class or named object declaration are satisfied; and
  - none of the field initializers initializes a field already initialized by some other declaration or extension of this object. **(this requires global knowledge; cut?)**

  As a result of the declaration, the typing context is extended with inheritance links from the class or object named *name* to each of the $superclass_i$ classes, and with subtyping relations from the class type corresponding to the class or object named *name* to the types corresponding to the $superclass_i$ classes.

- A predicate class declaration of the form

  `predicate` *name* **`isa`** $superclass_1$`, ...,` $superclass_M$ **`when`** *expr*`;`

  where `when` *expr* may be omitted, is type-correct iff:

  - *name* is not otherwise defined as a variable, class, or named object in the same scope;
  - each $superclass_i$ names a non-`void` class or named object;
  - no cycles are introduced into the inheritance graph; and
  - if the `when` clause is present,. *expr* is type-correct when checked in a typing context constructed by extending the current typing context with constant variable bindings for each of the $superclass_i$ with the type corresponding to the new predicate class (i.e., the greatest lower bound of the types corresponding to the predicate class's non-predicate ancestors), and returns an object of static type `bool`.

The typing context is extended to include a variable binding from *name* to a new predicate class that has the given `when` clause, and with inheritance links from the new predicate class to each of the $superclass_i$ classes.

- **disjoint and cover declarations**
- A synonym declaration of the form

  synonym *name* = *type*;

  is type-correct iff:

  - *name* is not otherwise defined as a type in the same scope; and
  - *type* notates a type in the current typing context, not involving (directly or indirect) *name*.

  The typing context is extended to include a type binding for *name* to *type*.

- A function declaration of the form

  fun *name* (x_1:$type_1$, ..., x_N:$type_N$) :$type_R$;

  is type-correct iff:

  - a function named *name* taking *N* arguments is not already defined in the same scope;
  - the x_i, when provided, are distinct;
  - each of the $type_i$, if provided, notates a non-`void` type in the current typing context, and otherwise is assumed to be `dynamic`; and
  - $type_R$, if provided, notates a type in the typing context, and otherwise is assumed to be `dynamic`.

  The typing context is extended to include a function binding for *name* taking *N* arguments, along with a signature of the form

  signature *name* ($type_1$, ..., $type_N$) :$type_R$

  (formal names are ignored).

- A signature declaration of the form

  signature *name* (x_1:$type_1$, ..., x_N:$type_N$) :$type_R$;

  is type-correct iff:

  - a function named *name* taking *N* arguments is already bound in the current typing context;
  - the x_i, when provided, are distinct;
  - each of the $type_i$, if provided, notates a non-`void` type in the current typing context, and otherwise is assumed to be `dynamic`; and
  - $type_R$, if provided, notates a type in the typing context, and otherwise is assumed to be `dynamic`.

  The typing context is extended to include the corresponding signature of the form

  signature *name* ($type_1$, ..., $type_N$) :$type_R$

  (formal names are ignored).

- A method declaration of the general form

  method *signature* *name* (..., x_i@c_i, ..., x_j:$type_j$, ...) :$type_R$ { *body* }

  having *N* formal parameters, where *signature* may be `signature` or empty and *body* may be code, a primitive, or a get or set field accessor, is type-correct iff:

  - a function named *name* taking *N* arguments is already bound in the current typing context;

- the x_i, when provided, are distinct;
- for each formal of the form x_i`@`c_i, c_i names a class or named object, and $type_i$ is considered to be the type corresponding to c_i (the type corresponding to a non-predicate class or named object is its corresponding class type, while the type corresponding to a predicate class is the greatest lower bound of the types corresponding to the predicate class's non-predicate ancestors);
- for every other formal, of the form x_j`:` $type_j$, if $type_j$ is provided, then it notates a non-`void` type in the typing context, otherwise $type_j$ defaults to `dynamic`;
- $type_R$, if provided, notates a type in the typing context, and otherwise is assumed to be `dynamic`;
- *body* is type-correct when checked in a typing context constructed by extending the current typing context with constant variable bindings for each of the x_i to the corresponding type $type_i$; and
- if $type_R$ is non-`void`, the type of the result of *body* is a subtype of $type_R$.

The following rules define type-correctness of the possible kinds of method bodies:

- A regular code method body of the form *body* is type-correct iff *body* is type-correct, and its result type is *body*'s result type.
- A primitive method body is always type-correct, with result type `dynamic`.
- The body of a get accessor field method without an initializer is always type-correct, and its result type is $type_R$.
- The body of a get accessor field method with an initializer of the form *body* is type-correct iff *body* is type-correct, and its result type is *body*'s result type.
- The body of a set accessor field method is always type-correct, and its result type is `void`.

The typing context is extended to include the declared method implementation. Moreover, if *signature* is `signature`, then the typing context is also extended to include the signature

`signature` *name* `(`$type_1$`,` ... `,` $type_N$`)` `:` $type_R$

(formal names are ignored).

- **precedence decls**
- Primitive declarations are always type-correct, and have no effect on the typing context.
- Pragma declarations are always type-correct, and have an implementation-dependent effect (usually none) on the typing context.

3.6 Mixed Statically- and Dynamically-Typed Code

One of Diesel's design goals is to support both exploratory programming and production programming and in particular to support the gradual evolution from programs written in an exploratory style to programs written in a production programming style. Both styles benefit from object-oriented programming, a pure object model, user-defined control structures using closures, and a flexible, interactive development environment. The primary distinction between the two programming styles relates to how much effort programmers want to put into polishing their systems. Programmers in the exploratory style want the system to allow them to experiment with partially-implemented and partially-conceived systems, with a minimum of work to construct and subsequently revamp systems; rapid feedback on incomplete and potentially inconsistent designs

is crucial. The production programmer, on the other hand, is concerned with building reliable, high-quality systems, and wants as much help from the system as possible in documenting and checking systems.

To partially support these two programming styles within the same language, type declarations and type checking are optional. Type declarations may be omitted for any argument, result, or local variable. Programs without explicit type declarations are smaller and less redundant, maximizing the exploratory programmer's ability to rapidly construct and modify programs. Later, as a program (or part of a program) matures, the programmer may add type declarations incrementally to evolve the system into a more documented and reliable production form.

Omitted type declarations are treated as `dynamic`; `dynamic` may also be specified explicitly as the type of some argument, result, or variable.* An expression of type `dynamic` may legally be passed as an argument, returned as a result, or assigned to a variable of any type. Similarly, an expression of any type may be assigned to, passed to, or returned from a variable, argument, or result, respectively, of type `dynamic`. This approach to integrating dynamically-typed code with statically-typed code has the effect of checking type safety statically wherever two statically-typed expressions interact (assuming that at run-time the objects resulting from evaluating the statically-typed expressions actually conform to the given types), and deferring to run-time checking at message sends whenever a dynamically-typed expression is used.

A consequence of this semantics for the `dynamic` type is that the static type safety of statically-typed expressions can be broken by passing an incorrect dynamically-typed value to a statically-typed piece of the program. Dynamic type checking will catch errors eventually, but run-time type errors can occur inside statically-typed code even if the code passes the type checker. An alternative approach would check types dynamically at the "interface" between dynamically- and statically-typed code: whenever a dynamically-typed value is assigned to (or passed to, or returned as) a statically-typed variable or result, the system could perform a run-time type check of the dynamically-typed value as part of the assignment. This approach would then ensure the integrity of statically-typed code: no run-time type errors can occur within statically-typed code labeled type-correct by the typechecker, even when mixed with buggy dynamically-typed code. Unfortunately, this approach has some difficulties. One problem is that objects defined in exploratory mode should not be required to include explicit subtyping declarations; such declarations could hinder the free-flowing nature of exploratory programming. However, if such an object were passed to statically-typed code, the run-time type check at the interface would fail, since the object had not been declared to be a subtype of the expected static type. We have chosen for the moment to skip the run-time check at the interface to statically-typed code in order to support use of statically-typed code from exploratory code, relying on dynamic checking at each message send to ensure that the dynamically-typed object supports all required operations.

Diesel supports the view that static type checking is a useful tool for programmers willing to add extra annotations to their programs, but that all static efficiently-decidable checking techniques are ultimately limited in power, and programmers should not be constrained by the inherent limitations

* In fact, the current UW Diesel implementation warns whenever an omitted type declaration defaults to `dynamic`.

of static type checking. The Diesel type system has been designed to be flexible and expressive (in particular by supporting multi-methods, separating the subtype and code inheritance graphs, and supporting explicit and implicit parameterization) so that many reasonable programs will successfully type-check statically, but we recognize that there may still be reasonable programs that either will be awkward to write in a statically-checkable way or will be difficult if not impossible to statically type-check in any form. Accordingly, error reports do not prevent the user from executing the suspect code; users are free to ignore any type checking errors reported by the system, relying instead of dynamic type checks. Static type checking is a useful tool, not a complete solution.

4 Parameterization and Bounded Parametric Polymorphism

Practical statically-typed languages need bounded parametric polymorphism. Without some mechanism for *type parameterization*, programmers must either resort to multiple similar implementations of the same abstraction that differ only in type annotations, or insert type casts, often at the client side, to indicate the more precise types of expressions than the type checker infers. For example, if parameterization is not available, several nearly identical implementations of `list` or `array` may be needed for lists or arrays of integers, strings, etc., and control structures such as `if` and `map` could not be reused for a variety of argument types. Accordingly, Diesel supports the definition of parameterized types and signatures, derived from parameterized classes, named objects, and functions. The programmer is allowed to express the assumptions on the type parameters in such declarations using mixed subtype and signature *type constraints*. For example, a type parameter may be restricted to be a subtype of a certain type or to be any type such that a certain signature holds. Type constraints in Diesel generalize F-bounded polymorphism [Canning et al. 89] and Theta-style `where` clauses [Day et al. 95, Liskov et al. 94].

This section presents type parameterization and type constraints in Diesel. A more formal treatment, in the context of Diesel's predecessor, Cecil, appears elsewhere [Litvinov 98, **Litvinov & Chambers TR**, **Litvinov thesis**]. The next subsection introduces parameterization. Subsection 4.2 adds constraints to achieve bounded parametric polymorphism. Subsection 4.6 describes constraint solving and local type inference. Subsection 4.5 describes an advanced use of the type system to express F-bounded polymorphism. The last subsection reviews related work.

4.1 Parameterized Declarations

Diesel supports parametric polymorphism by allowing many kinds of declarations to be parameterized with *type variables*. The most explicit way to make a declaration polymorphic is to prefix it with a `forall T1, ... ,Tn:` clause, which introduces type variables `T1, ... ,Tn` in a declaration. The scope of these type variables is the declaration that has this prefix, within which the type variables may be used as regular types; a type variable shadows any type of the same name declared in an enclosing scope. To use a polymorphic declaration, it first must be *instantiated* by providing (either explicitly or implicitly via a kind of local type inference) the *instantiating type* for each type variable: type variables are "formals" and instantiating types are "actuals" of a parameterized declaration. In the following example, an immutable vector class `i_vector`, a separate inheritance relationship, and a function `fetch` are each polymorphic in a given type variable (intended to denote the type of the vector elements):

```
forall T: class i_vector[T];
forall S: extend class i_vector[S] isa collection[S];
forall U: fun new_i_vector[U](len:int):i_vector[U] { .. new i_vector[U] .. }
forall R: fun fetch(a:i_vector[R], index:int):R { ... }
```

The basic syntax of parameterized declarations (ignoring some features described below) extends the earlier syntax of non-polymorphic Diesel in the following ways (changes to the right-hand-sides of previously presented non-terminals are highlighted, and the left-hand-sides of new non-terminals are highlighted):

```
class_decl      ::= [type_cxt] class_kind name [formal_params]
                        ["isa" class_refs] [field_inits] {pragma} ";"
ext_class_decl  ::= [type_cxt] "extend" ext_class_kind class_ref
                        ["isa" class_refs] [field_inits] {pragma} ";"
predicate_decl  ::= [type_cxt] "predicate" name [formal_params]
                        ["isa" class_refs] ["when" expr] {pragma} ";"
synonym_decl    ::= [type_cxt] "synonym" name [formal_params]
                        "=" type {pragma} ";"
fun_decl        ::= [type_cxt] "fun" fun_name [formal_params]
                        "(" [fun_formals] ")" [type_decl] {pragma} fun_body
method_decl     ::= [type_cxt] "method" ["signature"] fun_ref
                        "(" [meth_formals] ")" [type_decl] {pragma} method_body
signature_decl  ::= [type_cxt] "signature" fun_ref
                        "(" [fun_formals] ")" [type_decl] {pragma} ";"
field_decl      ::= [type_cxt] ["shared"] ["var"] "field" name [formal_params]
                        "(" fun_formal ")" [type_decl] {pragma} field_body
field_method_decl::=[type_cxt] ["shared"] ["var"] "field"
                        "method" ["signature"] name_fun_ref
                        "(" meth_formal ")" [type_decl] {pragma} field_body
type_cxt        ::= "forall" formal_param { "," formal_param } ":"
formal_params   ::= "[" formal_param { "," formal_param } "]"
formal_param    ::= name_binding
name_binding    ::= name
```

A type parameter is *explicit* if the corresponding instantiating type is to be explicitly provided by clients of the declaration, or *implicit* if it is to be inferred automatically by the typechecker. A polymorphic class, synonym, or function declaration specifies which type parameters are explicit by listing the corresponding type variables in brackets following the name of the declared entity, as in the `i_vector` class above. The explicitly instantiating types should be similarly given in brackets whenever the class, synonym, or function is referenced, as in the following example:

```
var my_vec:i_vector[num] := new_i_vector[num](10);
var result:num := fetch(my_vec, 5);
```

The `i_vector` class is declared with an explicit type parameter, `[T]`. Consequently, all uses of `i_vector` must provide an explicit instantiating actual type (`S` in the inheritance declaration, `U` in the `new_i_vector` function definition, `R` in the `fetch` function definition, and `num` in the `my_vec` variable declaration). Similarly, the `new_i_vector` function is declared with an explicit type parameter, `[U]`, and all uses of `new_i_vector` must provide an explicit instantiating actual type (`num` in the `new_i_vector` function call above). In contrast, the `fetch` function has no explicit type parameter; instead, the instantiating type for `R` is inferred at each use of `fetch` by the type checker itself. In the example call of `fetch`, `R` is inferred to be `num`, given the type of `my_vec`. Inference allows the programmer to avoid writing the often obvious instantiating types; it is a key feature of Diesel. It is described in more detail in Section 4.6.

The syntax of references to explicitly parameterized classes, objects, types, synonyms, and functions is extended as follows:

```
class_ref       ::= name [params]
name_fun_ref    ::= name [params]
op_fun_ref      ::= op_name [params]
```

```
params          ::= "[" types "]"
```

The number of explicit type parameters is considered part of the "name" of the declared entity. For example, multiple classes with the same name can be declared in the same scope, as long as they are declared with different numbers of explicit type parameters.* Likewise, multiple functions with the same name but different numbers of explicit type parameters can be declared in the same scope, analogously to how functions with the same name but different numbers of arguments can be declared in the same scope. Since signature, method, and field method declarations refer to previously declared functions, each must be declared with the same number of explicit type parameters as the function it is referencing. **explain somewhere the rules for ITC in the face of parameterized types. explain somewhere how overriding methods can add implicit type variables.**

While the type checker uses parameterized types to check polymorphic implementations and their clients, the execution behavior of a program does not depend on the instantiating types (aside from the number of explicit type parameters). In particular, method lookup does not examine any explicit or implicit type parameters for the function or its arguments. For example, it is not legal to define separate methods for `foo[int]()` and `foo[string]()` (as explained below in section **ITC**), and so message sends `foo[int]()` and `foo[string]()` will always be dispatched to the same method implementation.

Parameterized class extension declarations are only instantiated internally by the typechecker. Their type parameters therefore are always implicit.

put this in a better place: Note that parameterization is in general unsound and is disallowed in the following cases:

- Variable (`let`) declarations cannot be parameterized. (If parameterized variable declarations were allowed, to be sound, the initializing value, and each assigned value if the variable is mutable, would have to be a subtype of all possible instantiations of the variable's type.)
- The type of a field cannot reference any type parameters except those of the object to which the field is attached. Moreover, the type of a field attached to a named object cannot reference any type parameters **[really? why not? what should be done for a field on e.g. nil[T]? seems like there should be a cleaner way to talk about constraints of data vs. code, which subsumes the restriction on variables.]**.

4.2 Bounded Polymorphism and Type Constraints

It is often necessary to express some assumptions or restrictions on type parameters. For example, a `sort` method can only sort collections whose elements can be compared with each other. A `matrix_multiply` method may require that matrix elements be numbers. This situation is known as bounded polymorphism [Cardelli & Wegner 85]. Diesel supports bounded polymorphism by allowing *type constraints* on type parameters.

* This feature does not interact well with mixed dynamic and static typing, since the number of parameters affects the execution behavior of the program, violating the principle that static types do not affect the execution semantics. In the future, the number of parameters may be removed from the "name" of an object or method, so that parameters are confined to the (optional) static type system.

There are two kinds of type constraints in Diesel. A *subtype constraint* specifies the requirement that one type be a subtype of another. A common use of subtype constraints is to specify upper or lower bounds of type variables. In the following example, the type of matrix elements is constrained to be a subtype of `num`:

```
forall T where T<=num:
   fun matrix_multiply(a:matrix[T], b:matrix[T]):matrix[T] {
      ...
   }
```

Because of the type constraint, this function can only be instantiated for types that are subtypes of `num`.

A *signature constraint* specifies the requirement that the given signature hold. A common use of signature constraints is to require certain operations to be provided for the type parameters. In the following example, the message send of <= in the body of `sort` is guaranteed to be legal as long as the constraint is satisfied:

```
forall T where signature <=(:T,:T):bool:
   fun sort(a:array[T]):void {
      ...
      let a_i:T := a!i;
      let a_j:T := a!j;
      if(a_i <= a_j, { ... swap a!i and a!j ... });
      ...
   }
```

Type constraints can be specified as part of `forall` clauses:

```
type_cxt       ::= "forall" formal_param { "," formal_param } [type_cons] ":"
                 |   "forall" type_cons ":"
```

The syntax of type constraints is as follows:

```
type_cons      ::= "where" type_constraint { "," type_constraint }
type_constraint::= sub_constraint | sig_constraint | type
sub_constraint ::= type ("<=" | ">=") type
sig_constraint ::= ["signature"] fun_ref "(" [fun_formals] ")" type_decl
```

As a convenience, a single lower and/or upper bound may also be specified with a type variable when it is introduced:

```
name_binding   ::= name [">=" type] ["<=" type]
```

For example, the `matrix_multiply` function above could have been declared equivalently as:

```
forall T<=num:
   fun matrix_multiply(a:matrix[T], b:matrix[T]):matrix[T] {
      ...
   }
```

The specializer of a method's formal can be written as a type variable whose upper bound is the specializing class:

```
meth_formal    ::= [name] ":" type
                 |   [name] "@" [name "<="] class_ref
                 |   name
```

For example, in the following method:

```
forall T: method signature foo(x@T<=bar):T { ... }
```

the specializer is the class `bar`, and so this method applies to any objects that inherit from `bar`, but moreover, because of the type parameter, it is known that the result will be of the same type as the argument, which can be more precise than just `bar`. In this example, the following signature is derived:

```
forall T<=bar: signature foo(x:T):T;
```

4.3 Omitting the Explicit `forall` Clause: the Backquote Sugar

Type variables can be introduced implicitly — without listing them in the `forall` clause. This provides a more concise notation for parameterized declarations by omitting the explicit `forall` prefix. A type variable is introduced implicitly in a declaration iff:

- it is an explicit formal type parameter, i.e., a type introduced in square brackets following the name of a class, object, synonym, function, signature, or method being declared; or
- it is preceded by a backquote (`` ` ``) somewhere in the declaration's header.

So the declarations from section 4.1 can be rewritten more concisely as:

```
class i_vector[T];
extend class i_vector[`T] isa collection[T];
fun fetch(a:i_vector[`T], index:int):T  { ... }
```

To specify type constraints without requiring explicit `forall` clauses, an upper and/or lower bound may be specified along with a backquoted type variable, and every declaration allowing a `forall` clause also allows a `where` clause to be provided at the end of its header. For example, the `matrix_multiply` and `sort` functions from section 4.2 can be re-written more concisely as follows (the `matrix_multiply` example sacrifices the visual symmetry between the arguments `a` and `b`, but is semantically equivalent, because it introduces exactly the same type variable and constraint):

```
fun matrix_multiply(a:matrix[`T<=num], b:matrix[T]):matrix[T] { ... }
fun sort(a:array[T]):void where signature <=(:T,:T):bool {
   ...
   let a_i:T := a!i;
   let a_j:T := a!j;
   if(a_i <= a_j, { ... swap a!i and a!j ... });
   ...
}
```

Similarly, the operations on a binary tree can specify that the elements of the binary tree support the comparison operation:

```
class binary_tree[T] isa collection[T];
fun insert(t:binary_tree[`T], elm:T):void where signature <=(:T,:T):bool {
   ... }
fun includes(t:binary_tree[`T], elm:T):bool where signature <=(:T,:T):bool {
   ... }
```

In this case, only instantiations of `binary_tree` with comparable element types will be useful in practice, since all non-trivial uses will involve a function requiring ordered elements, such as `insert` and `includes` above. To capture this constraint, the `binary_tree` class itself can be given the constraint:

```
class binary_tree[T] where signature <=(:T,:T):bool;
```

This ensures that every instantiation of `binary_tree` has an element type that can be compared. As described so far, however, the type constraints on the `insert` and `includes` functions are still necessary, in order to satisfy the constraints of the instantiation of `binary_tree`. However, since these binary trees are arguments, instantiated previously, it is known that they must have satisfied all the required type constraints of `binary_tree` when the binary tree instances were created. To avoid manually writing down such "known" type constraints, Diesel provides a syntactic sugar that automatically inserts these associated constraints. If a backquoted type variable is used as an explicit instantiating parameter of a parameterized type, the constraints that the type associates with its explicit parameter in the corresponding position are imposed on the type variable. In the `insert` and `includes` functions, the occurrence of `'T` implicitly adds to the enclosing function declaration the type constraints required for `T` to be a legal instance of `binary_tree`, i.e., `signature <=(:T,:T):bool`. Thus, these functions can be declared simply as follows:

```
fun insert(t:binary_tree['T], elm:T):void { ... }
fun includes(t:binary_tree['T], elm:T):bool { ... }
```

Aside defining locations where constraints are inferred, there is no semantic impact from marking more than one occurrence of a type variable with a backquote; any one occurrence is enough to ensure that the type variable is included in the declaration's (implicit or explicit) `forall` clause. Similarly, it does not matter which occurrences of a backquoted type variable have upper and/or lower bounds specified; all will be promoted tothe declaration's `forall` clause.

The full syntax of parameterized types, specifying formally where backquoted type variables are allowed, is as follows:

```
class_decl     ::= [type_cxt] class_kind name [formal_params]
                       [type_cons] ["isa" class_ref_ps] [field_inits]
                       {pragma} ";"
class_ref_ps   ::= class_ref_p { "," class_ref_p }
class_ref_p    ::= name [param_patterns]
ext_class_decl ::= [type_cxt] "extend" ext_class_kind class_ref_p
                       [type_cons] ["isa" class_ref_ps] [field_inits]
                       {pragma} ";"
predicate_decl ::= [type_cxt] "predicate" name [formal_params]
                       [type_cons] ["isa" class_ref_ps]
                       ["when" expr] {pragma} ";"
synonym_decl   ::= [type_cxt] "synonym" name [formal_params]
                       "=" type_pattern [type_cons] {pragma} ";"
fun_decl       ::= [type_cxt] "fun" fun_name [formal_params]
                       "(" [fun_formals] ")" [type_decl_p] [type_cons]
                       {pragma} fun_body
fun_formal     ::= [name] ":" type_pattern
                 |   name
```

```
method_decl     ::= [type_cxt] "method" ["signature"] formal_fun_ref
                        "(" [meth_formals] ")" [type_decl_p] [type_cons]
                        {pragma} method_body
formal_fun_ref  ::= name_formal_fun_ref | op_formal_fun_ref
name_formal_fun_ref::=name [formal_params]
op_formal_fun_ref::=op_name [formal_params]
meth_formal     ::= [name] ":" type_pattern
                |   [name] "@" ["`"] name "<="] class_ref_p
                |   name
signature_decl  ::= [type_cxt] "signature" formal_fun_ref
                        "(" [fun_formals] ")" [type_decl_p] [type_cons]
                        {pragma} ";"
field_decl      ::= [type_cxt] ["shared"] ["var"] "field"
                        name [formal_params]
                        "(" fun_formal ")" [type_decl_p] [type_cons]
                        {pragma} field_body
field_method_decl::=[type_cxt] ["shared"] ["var"] "field"
                        "method" ["signature"] name_formal_fun_ref
                        "(" meth_formal ")" [type_decl_p] [type_cons]
                        {pragma} field_body
type_constraint::= sub_constraint | sig_constraint | type_pattern
sub_constraint  ::= type_pattern ("<=" | ">=") type_pattern
sig_constraint  ::= ["signature"] formal_fun_ref
                        "(" [fun_formals] ")" type_decl_p
type_decl_p     ::= ":" type_pattern
type_patterns   ::= type_pattern { "," type_pattern }
type_pattern    ::= lub_type_p
lub_type_p      ::= lub_type_p "|" glb_type_p
                |   glb_type_p
glb_type_p      ::= glb_type_p "&" simple_type_p
                |   simple_type_p
simple_type_p   ::= binding_type_p
                |   named_type_p
                |   closure_type_p
                |   "(" type_pattern ")"
binding_type_p  ::= "`" name_binding
named_type_p    ::= class_ref_p
closure_type_p  ::= "&" "(" [arg_type_ps] ")" [type_decl_p]
arg_type_ps     ::= arg_type_p { "," arg_type_p }
arg_type_p      ::= [[name] ":"] type_pattern
formal_param    ::= ["`"] name_binding
name_binding    ::= name [">=" type_pattern] ["<=" type_pattern]
```

4.4 Polymorphism and Subtyping

discuss non-, co-, and contravariant type parameters, e.g.:

by default, there is no subtyping relation between two different instances of the same parameterized type. however, sometimes such a relationship is desired. this can be expressed using constrained implicit type parameters. e.g.:

```
class indexed[T] isa indexed[`S >= T]; -- subtyping of indexed is covariant in T
   fun fetch(i:indexed[`T], index:int):T { ... }
```

```
class m_indexed[T] isa indexed[T]; -- subtyping of m_indexed is nonvariant in T
   fun store(i:indexed['T], index:int, val:T):void { ... }
-- some allowed and disallowed subtyping relations:
m_indexed[int] <= indexed[int] <= indexed[num] <= indexed[any]
m_indexed[int] !<= m_indexed[num] -- because it would be unsound
```

here indexed[T] is a read-only collection of elements of type T. indexed[T] can be legally viewed as a subtype of indexed[S] for any supertype S of T, i.e., the subtyping of indexed is covariant in its type parameter. in contrast, m_indexed[T] is a read-write collection of elements of type T. in this situation, it would be unsound in general to view an m_indexed[T] as an m_indexed[S] for any type S != T. E.g.:

```
let a1:m_indexed[int] := new m_indexed[int];
let a2:m_indexed[any] := a1; -- (not allowed in real Diesel)
a2.store(0, "hello");
let x:int := a1.fetch(0); -- expects to return an int, but gets a string instead
```

in this case, the subtyping of m_indexed is non-variant in its type parameter, which is the default situation. an m_indexed[T] is a subtype of indexed[T], which is a subtype of indexed[S] for any S >= T, so by transitivity m_indexed[T] is a subtype of indexed[S] for any S >= T. this does not cause problems because the view of the mutable collection through indexed[S] does not allow any updates, only reads, thereby avoiding the soundness problem.

subtyping of a type may also be contravariant in a type parameter. for example, subtyping of closure types is contravariant in the argument types (and covariant in the result type). e.g. a user-defined type analogous to a two-argument closure type could be declared as follows:

```
abstract class Fun[Arg1, Arg2, Result] -- equivalent to &(Arg1,Arg2):Result
      isa Fun['Arg1B <= Arg1, 'Arg2B <= Arg2, 'ResultB >= Result];
   fun eval(f:Fun['Arg1,'Arg2,'Result], a1:Arg1, a2:Arg2):Result;
-- some allowed and disallowed subtyping relations:
Fun[any,any,none] <= ... <= Fun[num,any,int] <= ... <= Fun[int,string,num]
<= ... <= Fun[none,none,any]
```

discuss conditional subtyping, e.g.

```
abstract class printable;
   fun print(p:printable):void;
abstract class collection[T];
extend class collection['T <= printable] isa printable;
   method print(a@collection['T <= printable]):void {
      print("["); a.do(&(e:T){ print(e); }); print("]"); }
```

whether a function is in a type's (implicit) interface can be conditional on properties of the type's instantiating type parameters, by adding type constraints to the function's definition. e.g. matrix_multiply and sort above. in other words, a function may be defined only for certain instantiations of its argument types.

similarly, only certain instances of a polymorphic type may subtype from another. this conditional subtyping arises from class extension declarations that impose more type constraints than the class being extended. inheritance is always there, but the functions in the interface of the inherited class

are only invocable for instances of the subclass that satisfy the additional type constraints. e.g. print only supported for collections of printable elements.

4.5 F-bounded Polymorphism

This subsection describes an example of advanced use of the Diesel type system, F-bounded polymorphism. As we will see, no special support for this powerful idiom is needed in the type system — it is made possible by allowing constraints to be recursive, whereby a type variable can appear in its own bound.

For our first example, let us consider an abstract class `ordered` and a binary function <=. A *binary function* is a function that expects two arguments of similar types; the <= function can be applied, for example, to two numbers or two strings, but not a string and a number. We would like to define this function once, in the `ordered` class, and have other classes, such as `num` and `string`, inherit it. The simplest way to achieve it seems to be as follows:

```
abstract class ordered;
   fun <=(x:ordered, y:ordered):bool;
   fun > (x:ordered, y:ordered):bool { not(x <= y) }

extend class num isa ordered;
extend class string isa ordered;
```

This code, however, leads to an undesirable effect. Since <= and > are defined for `ordered` and `num` and `string` are its subclasses, we are required (for completeness) to write implementations of <= to compare a `num` and a `string`, which we may not want. To avoid mixing of subclasses of `ordered`, we can apply F-bounded polymorphism as follows:

```
abstract class ordered[T] where T <= ordered[T];
   fun <=(x:'T, y:'T):bool where T <= ordered[T];
   fun > (x:'T, y:'T):bool where T <= ordered[T] { not(x <= y) }

extend class num isa ordered[num];
   method <=(x@num, y@num):bool { ... }

extend class string isa ordered[string];
   method <=(x@string, y@string):bool { ... }
```

Now function > can be instantiated with `num` for `T` (because the instantiated constraint `num <= ordered[num]` can be solved: there is a corresponding declaration in the program) or with `string` for `T`, but cannot with `(string|num)` for `T` (which would be required in order to compare a `num` and a `string`).

With this scheme, in addition to defining binary functions itself, `ordered` and all its subtypes can inherit binary functions from other objects, for example:

```
abstract class comparable[T] where T <= comparable[T];
   fun = (x:'T, y:'T):bool where T <= comparable[T];
   fun !=(x:'T, y:'T):bool where T <= comparable[T] { not(x = y) }

extend class ordered['T] isa comparable[T];
   method =(x@num, y@num):bool { ... }
   method =(x@string, y@string):bool { ... }
```

Moreover, `num` can have subtypes, such as `int` or `float`, which can be compared with each other, but not with `string` or its subtypes:

```
extend class int isa num;
extend class float isa num;
3 != 3.14    -- legal
```

F-bounded polymorphism can be applied similarly to express families of two or more mutually recursive types. For example, consider a simplified model-view (also known as subject-observer) framework, where the model and the view must be able refer to each other and invoke operations on each other.* Moreover, instances of the model-view framework, such as a drawing model and a drawing view, must be able to invoke specific operations on each other without loss of static type safety. The following code shows how the generic model-view framework can be defined:

```
abstract class model[`M <= model[M,V], `V <= view[M,V]];
   field views(:model[`M,`V]):set[V] := new_set[V]();
   fun register_view(m:model[`M,`V], view:V):void {
      m.views.add(view); }
   fun changed(m:model[`M,`V]):void {
      m.views.do(&(v:V){
         v.update();
      }); }

abstract class view[`M <= model[M,V], `V <= view[M,V]];
   field model(:view[`M,`V]):M;
   fun update(v:view[`M,`V]):void;
```

Both `model` and `view` are parameterized by the type of the model and the view with the corresponding upper bounds on these two parameters. Correspondingly, the code for the `model` and `view` classes is parameterized by the actual types of the instantiation of the framework. For example, the following code instantiates the generic model-view framework to construct a bitmap drawing model and view:

```
class drawing isa model[drawing,drawing_view];
   field bitmap(:drawing):bitmap;
   method set_pixel(m:drawing, pos:position, value:color):void {
      m.bitmap.pixel(pos) := value;
      -- more efficient than simply m.changed():
      m.views.do(&(v:drawing_view){
         v.update_pixel(pos, value);
      }); }
   fun new_drawing():drawing { new drawing }

class drawing_view isa view[drawing,drawing_view];
   method update(v@drawing_view):void {
      screen.plot(v.model.bitmap); }
   fun update_pixel(v:drawing_view, pos:position, value:color):void {
      screen.plot_pixel(pos, value); }
```

* Thanks to Gail Murphy for suggesting this problem to us.

```
fun new_drawing_view(m:drawing):drawing_view {
   new drawing_view { model := m } }
```

Both `drawing` and `drawing_view` add new operations that need to be called by the other type. By parameterizing `model` as was done, the type of the `views` field in `drawing` is known statically to be set of (subtypes of) `drawing_view`. This knowledge allows the `set_pixel` operation in `drawing` to invoke the `update_pixel` operation without generating either a static type-error or requiring a dynamic "typecase" or "narrow" operation. Similarly, because of the way `view` is parameterized, the `model` field in its child `drawing_view` will be known statically to refer to a (subtype of) `drawing`, allowing the `update` operation of `drawing_view` to access the `bitmap` field of the model in a statically type-safe manner. Note that it is legal to instantiate `model` and `view` with `drawing` and `drawing_view`, because the instantiated subtype constraints can be solved successfully.

Alternatively to the unparameterized `drawing` and `drawing_view`, the programmer could parameterize them in a way similar to how `model` and `view` are parameterized, in order to allow further refinement of these two types. This is similar to having the parameterized `ordered` subtype of `comparable`, as opposed to the unparameterized `num` and `string` subtypes of `ordered`, in our earlier examples.

4.6 Constraint Solving and Local Type Inference

The following typechecking tasks in Diesel lead to *constraint solving*:

- To typecheck a message send $m[T_1',...,T_m'](E_1,...,E_n)$, where the types of $E_1,...,E_n$ are $T_1,...,T_n$, the signature constraint `signature` $m[T_1',...,T_m']$ $(T_1,...,T_n)$:$\mathtt{T}_{result}$ is solved. Here $\mathtt{T}_{result}$ is a fresh type variable and can be instantiated with some type. The type of the message send is the most specific type that $\mathtt{T}_{result}$ can take on while the signature constraint can be solved successfully.
- Type-checking many other kinds of expressions, statements, and declarations involves determining whether one type is a subtype of another. To check whether S_1 is a subtype of S_2, the subtype constraint $S_1 <= S_2$ is solved. S_1 is a subtype of S_2 iff the constraint can be solved successfully.
- Whenever a declaration with constraints in its header is instantiated, the instantiated constraints must be solved. If they cannot be solved successfully, such instantiation is not *legal* and so is disallowed.

Informally, given a constraint to solve, constraint solving proceeds as follows. A "set-to-be-solved" of constraints is created, initially containing this one constraint. One constraint at a time is picked and removed from this set. A matching constraint is produced from the program declarations, if possible, otherwise constraint solving fails. Two constraints match if they have the same structure (*e.g.*, both are signature constraints for the same function) and the types in the corresponding positions are the same; fresh type variables may be instantiated with types during matching. While producing the matching constraint, new constraints to be solved may arise, in which case they are added to the set-to-be-solved. Constraint solving succeeds when the set-to-be-solved becomes empty.

The matching constraint can be produced either by taking a constraint or declaration available in the program, or by combining other constraints produced from the program declarations. More specifically:

- A polymorphic subtype or signature declaration present in the program can be instantiated by substituting types or fresh type variables for its type variables; its constraints, if any, need to be solved and are added to the set-to-be-solved. A subtype of signature declaration with no type variables is treated as an available constraint itself.
- When typechecking the body of a polymorphic declaration, the constraints in its header are available.
- Constraints can be combined based on the standard properties of subtyping, such as transitivity, and of signatures, such as contravariance. For example, if the program contains declarations `signature=(:num,:num):bool` and `extend class int isa num`, they can be combined to yield the constraint `signature=(:int,:num):bool`. Matching of types and substitutions of types for fresh type variables are performed as needed.

Inference of instantiating types is the part of constraint solving whereby the typechecker decides how to instantiate polymorphic declarations, i.e., what types to substitute for type variables. Intuitively, when typechecking a message send, the typechecker tries to find the "best" instantiations of declarations involved in solving the signature constraint, i.e., the instantiations that lead to the most precise result type. When checking whether a type is a subtype of another, the typechecker only needs to prove that some appropriate instantiations exist.

Consider, for example, typechecking the message send `print(my_coll)` in the context of the following declarations:

```
abstract class printable;
fun print(p:printable):void;


abstract class collection[T];
extend class collection[`T <= printable] isa printable;
method print(a@collection[`T <= printable]):void {
   print("["); a.do(&(e:T){ print(e); }); print("]"); }


class frob isa printable;
method print(a@frob):void { ... }


let my_coll:collection[frob] := ...;
print(my_coll);
```

Since `my_coll` has type `collection[frob]`, in order to check this send, the typechecker needs to solve the constraint `signature print(collection[frob]):`$\mathtt{T}_{result}$ where $\mathtt{T}_{result}$ is a fresh type variable; this is the first constraint in the set-to-be-solved. There is only one declaration available that can be matched against this constraint: the one for `signature print(printable):void`, derived from the function declaration. Using the implicit contravariant signature matching rule, the original constraint can be satisfied if we can satisfy the following new constraints: `collection[frob] <= printable` and `void <=` $\mathtt{T}_{result}$. The latter constraint can be satisfied by instantiating $\mathtt{T}_{result}$ with any supertype of `void`; we wish the result of the

message to be as precise as possible, so our inferencer picks the most specific legal instantiation, in this case `void`. The former constraint can be satisfied by matching against the conditional subtyping declaration `collection['T<=printable] <= printable`, if we are able to satisfy match `frob` against `'T<=printable`. This matching succeeds if we can substitute `frob` for `T` and also satisfy the constraints on `T`. This leads to needing to solve the constraint `frob <= printable`, which is directly satisfied by the subtyping knowledge in `frob`'s class declaration. All the constraints are satisfied, and so the message send is type-correct, returning a value of type `void`.

explain somewhere the rules for ITC in the face of parameterized types. generally, revisit typechecking rules from section 3, revising/augmenting to account for type parameters.

A fuller and more formal explanation of Diesel's constraint-based type system and the constraint solving algorithm, along with soundness proofs and a discussion of termination, appear elsewhere [Litvinov 98, **Litvinov & Chambers TR**, **Litvinov thesis**].

4.7 Related Work

THIS SUBSECTION STILL NEEDS TO BE UPDATED

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, languages based on covariant redefinition, and languages offering local type inference. Diesel (and its predecessor Cecil) 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 Diesel, although sometimes with more compact syntax. Also, the other languages do not support multi-methods or least-upper-bound and greatest-lower-bound type expressions, except where noted below.

4.7.1 Languages Based on F-Bounded Polymorphism

Pizza is an extension to Java based on F-bounded polymorphism [Odersky & Wadler 97]. Like Diesel, Pizza supports classes with mutually recursive bounds, crucial for supporting interrelated families of classes such as the `model-view` example from section 4.5. Also like Diesel, 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 Diesel which can infer appropriate supertypes of the argument types. 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 our `i_vector` example. 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. Diesel's multi-methods 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 `model-view` example from section 4.5 illustrates our approach. 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 are 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 Diesel in being able to parameterize a class over its superclass. However, this feature cannot be typechecked 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; Diesel 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]. $ML_{\leq}$ supports subtyping directly, but treats inheritance as a separate syntactic sugar (which must follow the subtyping relation). Similarly to Diesel, $ML_{\leq}$ constrains type variables using sets of potentially recursive subtype constraints, supports inference of type parameters to methods, and supports least-upper-bound type expressions (although not greatest-lower-bound type expressions). $ML_{\leq}$ also supports parameterization over type constructors, while in Diesel type constructors must be instantiated before use. $ML_{\leq}$ supports explicit declarations of co- and contravariant type parameters of type constructors, while Diesel uses polymorphic subtype declarations to achieve more general effects. $ML_{\leq}$ only allows 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 $ML_{\leq}$ does not support other forms of constrained subtyping, conformance, or inheritance. Diesel supports multiple polymorphic signature declarations for the same message, while $ML_{\leq}$ allows only a single signature declaration per message. $ML_{\leq}$ is purely functional and side-effect-free.

4.7.2 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 <= C[SelfType]:
 template object C[SelfType];
```

`SelfType` supports binary methods like <= 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 with 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. 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].

discuss Diesel's unification of inheritance & subtyping vs. Cecil, and how F-bounded compensates for much of that lost expressiveness.

4.7.3 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 subtyping relations that ignore behavioral specifications, and may interact better with inheriting default implementations of protocol types. Neither is clearly better than the other; Diesel supports both easily. In addition, Diesel allows new supertypes to be added to previously declared types, avoiding one limitation of explicit subtyping when adding new explicit protocol types and adapting previously written objects 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 Diesel, 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 Diesel, type declarations and typechecking are 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 Diesel's type system.

Theta [Day et al. 95, Liskov et al. 94] and PolyJ [Myers et al. 97] support signature constraints called `where` clauses. Unlike Diesel, 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 Diesel must infer a unique, most-specific ground type. As a result, their system can typecheck programs Diesel cannot. On the other hand, because Diesel 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 least-upper-bound and greatest-lower-bound subtype constraints can appear to ensure that such constraints can always be solved, while Diesel places no syntactic limits but may report a type error due to incompleteness of the particular deterministic algorithm used by the typechecker.

4.7.4 Languages Based on Instantiation-Time Checking

Some languages, including C++ [Stroustrup 86] 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.

4.7.5 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-Pedersen 89, Madsen et al. 93], and Thorup adapted this idea in his proposed virtual types extension to Java [Thorup 97]. 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, Diesel can directly support all of the standard examples used to justify such mechanisms (including binary methods and the `model-view` 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.

4.7.6 Languages Offering Local Type Inference

The work on local type inference in an extension of $F_{\leq}$ [Pierce & Turner 98], especially the "local type argument synthesis," is very similar to inference of instantiating types in Diesel: they address a similar problem and use a similar inference algorithm. Their setting is different from Diesel's: they work within an impredicative type system whereas Diesel's is essentially predicative. In contrast with their system, Diesel handles F-bounded quantification, signature constraints, by-name subtyping, and overloading (with multiple dispatch). An earlier work on type inference in $F_{\leq}$ [Cardelli 93] presents a faster algorithm which is more restrictive in some cases due to asymmetric treatment of method arguments.

A similar kind of type inference is also offered by GJ, a language that adds parameterized types to Java [Bracha et al. 98]. Compared to its predecessor, Pizza, in GJ the type of an expression does not depend on its context, and the type inference supports subsumption and empty collections (which may be considered as having multiple incomparable collection types). GJ only provides non-variant type parameters whereas in Diesel covariant or contravariant type parameters can be expressed using polymorphic subtype declarations and are supported by type inference. Type inference in GJ seeks to find the smallest instantiating types for type variables, whereas the goal of

type inference in Diesel is to infer the most specific type of an expression (which may be achieved, for example, with the biggest instantiating type for a contravariant type parameter). GJ supports F-bounded polymorphism, but does not provide other advanced language constructs, such as signature constraints, independently parameterized subtype declarations, and multi-methods. The authors of GJ report on the positive experience with their 20,000-line GJ compiler (written in GJ, too) which extensively uses parameterization for container classes and the Visitor pattern. The 125,000-line Vortex compiler written in Diesel [Dean et al. 96] also uses parameterization extensively for container classes as well as in heavily parameterized optimization and interprocedural analysis frameworks [Litvinov 98]. Since Diesel allows additions of new multi-methods and new branches of multi-methods to the existing code, there is no need to use the Visitor pattern in Vortex.

5 Modules

Diesel allows declarations to be packaged up into modules. Modules aid in managing a program's name spaces, by treating like-named declarations in different modules as unrelated. Client code explicitly indicates which subset of the available modules are being used, thereby helping to narrow the interface that the client has of the rest of the program. Modules also support encapsulation, by allowing the declarations within a module to be marked with a visibility annotation, and restricting clients to access only visible members. The following example illustrates some of the features of Diesel's module system:

```
module Collection {
   import PrintUtils; -- import e.g. print_to_console
   public abstract class collection[T];
   public fun do(c:collection[`T], closure:&(T):void):void;
   public fun print(c:collection[`T]):void {
      print_to_console(c.collection_name);
      ... print elements ... }
   protected fun collection_name(:collection[T]):string;
   ...
}
module List {
   public extends Collection;
   public class list[T] isa collection[T];
   method collection_name(l@list[`T]):string { "list" }
   import Link;
   module Link {
      public class link[T];
         public field value(:link[T]):T;
         public get var field next(:link[T]):link[T]|null;
      public object null;
      ...
   }
   ...
}
```

The full syntax of module-related declarations augments the earlier Diesel syntax as follows:

```
static_decl    ::= module_decl
                |  ext_module_decl
                |  import_decl
                |  extends_decl
                |  class_decl
                |  ext_class_decl
                |  predicate_decl
                |  disjoint_decl
                |  cover_decl
                |  divide_decl
                |  synonym_decl
                |  fun_decl
                |  method_decl
                |  signature_decl
```

```
                |   field_decl
                |   field_method_decl
                |   precedence_decl
                |   prim_decl
                |   pragma
module_decl     ::= [privacy] "module" name {pragma} module_body
module_body     ::= "{" module_contents "}" [";"]
                |   ";" module_contents "end" "module" [name] ";"
                |   ";" module_contents <EOF>     module body ends at end of file
module_contents::= { mod_decl | stmt }
mod_decl        ::= friend_decl
                |   static_decl
                |   dyn_decl
ext_module_decl::= "extend" module_decl
import_decl     ::= [privacy] "import" module_refs {pragma} ";"
extends_decl    ::= [privacy] "extends" module_refs {pragma} ";"
friend_decl     ::= "friend" module_refs {pragma} ";"
module_refs     ::= module_ref {"," module_ref}
module_ref      ::= qualified_name
class_decl      ::= [type_cxt] [privacy] class_kind name [formal_params]
                        [type_cons] ["isa" class_ref_ps] [field_inits]
                        {pragma} ";"
ext_class_decl  ::= [type_cxt] [privacy] "extend" ext_class_kind class_ref_p
                        [type_cons] ["isa" class_ref_ps] [field_inits]
                        {pragma} ";"
predicate_decl  ::= [type_cxt] [privacy] "predicate" name [formal_params]
                        [type_cons] ["isa" class_ref_ps]
                        ["when" expr] {pragma} ";"
synonym_decl    ::= [type_cxt] [privacy] "synonym" name [formal_params]
                        "=" type_pattern [type_cons] {pragma} ";"
fun_decl        ::= [type_cxt] [privacy] "fun" fun_name [formal_params]
                        "(" [fun_formals] ")" [type_decl_p] [type_cons]
                        {pragma} fun_body
method_decl     ::= [type_cxt] [privacy] "method" ["signature"] formal_fun_ref
                        "(" [meth_formals] ")" [type_decl_p] [type_cons]
                        {pragma} method_body
signature_decl  ::= [type_cxt] [privacy] "signature" formal_fun_ref
                        "(" [fun_formals] ")" [type_decl_p] [type_cons]
                        {pragma} ";"
field_decl      ::= [type_cxt] [field_privacy] ["shared"] ["var"] "field"
                        name [formal_params]
                        "(" fun_formal ")" [type_decl_p] [type_cons]
                        {pragma} field_body
field_method_decl::=[type_cxt] [field_privacy] ["shared"] ["var"] "field"
                        "method" ["signature"] name_formal_fun_ref
                        "(" meth_formal ")" [type_decl_p] [type_cons]
                        {pragma} field_body
let_decl        ::= [privacy] "let" ["var"] name [type_decl] {pragma}
                        ":=" expr ";"
privacy         ::= "public" | "protected" | "private"
field_privacy   ::= privacy ["get" [privacy "put"] | "put"]
var_ref         ::= qualified_name
class_ref       ::= qualified_name [params]
class_ref_p     ::= qualified_name [param_patterns]
```

```
name_fun_ref    ::= qualified_name [params]
op_fun_ref      ::= qualified_op_name [params]
name_formal_fun_ref::=qualified_name [formal_params]
op_formal_fun_ref::=qualified_op_name [formal_params]
qualified_name ::= [[module_ref] "$"] name
qualified_op_name::=[[module_ref] "$"] op_name
```

Note that Diesel's current module system is intentionally simple and somewhat incomplete, intended to enable Diesel programmers (and its language designer) to gain experience using a module system in a language featuring Diesel's other unusual constructs such as multiple dispatching.

5.1 Module Declarations

A group of declarations and statements can be packaged in a named module. The module introduces a new nested scope, in which its body declarations are declared. Executing a module causes its body statements and declarations to be executed, in the order given in the module. As with other nested scoping, declarations in the scope surrounding the module are visible within the module. (Visibility of the declarations within a module to scopes outside the module are mediated by privacy annotations, as discussed in section 5.2.)

There are several equivalent ways of declaring a module:

module *moduleName* { ... *decls and stmts* ... }
module *moduleName;* ... *decls and stmts* ... **end module;**
module *moduleName;* ... *decls and stmts* ... **end module** *moduleName***;**
module *moduleName;* ... *decls and stmts* ... -- *end of file*

The first way is the "standard" way to write a nested scope, but if the body is large, the closing brace may be visually difficult to notice. As a more visually obvious alternative, the `end module`-style syntax can be used. For further clarity and error checking, the name of the module can be specified again when it is being ended. Finally, the `end module` clause can be omitted entirely, in which case the module implicitly extends to the end of the file; this style is particularly convenient if a whole file defines a single module.

A module can be declared within another module, introducing a further nested scope. As with nested scoping in other contexts, the declarations of the enclosing module are visible within the nested module.

File `include` declarations are only allowed at top-level, not within a module. This ensures that an included file's scope always begins in the outer scope.

5.2 Privacy and Encapsulation

Most kinds of declarations in a module can be given privacy annotations, which control the accessibility of the declaration in scopes outside the module. (A declaration is always accessible within the module and within any nested modules. As a consequence, it is pointless to give privacy annotations on global declarations.) If a named declaration is accessible, then it can be referenced,

either directly using a qualified name (described in section 5.3) or indirectly through the effects of an `import` declaration (described in section 5.4).

A declaration annotated `public` is accessible to any scope that has access to the enclosing module. Since global modules are accessible to all code, public declarations of global modules are globally accessible.

A declaration annotated `protected` is accessible to outside code only if it is lexically within a module that declares itself to be a "submodule" of the module directly containing the `protected` declaration. One module declares that it is a submodule of another using an `extends` declaration, as described in section 5.5.

A declaration annotated `private` is accessible to outside code only if it is lexically within a module that the module directly containing the `private` declaration declares to be a friend, using a `friend` declaration as described in section 5.6.

If a privacy annotation is omitted, it defaults to `protected`.

A mutable field declaration introduced two new names, one for the field's getter function and one for its setter function. The privacy of each of these functions can be specified separately. A field privacy annotation of the form *getter_privacy* `get` *setter_privacy* `put` specifies both the getter and the setter's privacy explicitly. Either of these halves may be omitted, in which case the omitted privacy defaults to `protected`. A single privacy annotation can be given, in which case it applies to both the getter and the setter.

For all declarations that introduce one or more new names, including module, class, object, predicate, synonym, function, field, and variable declarations, privacy annotations control accessibility of those names. Some declarations do not introduce any new names, but rather augment some existing declared entities; such declarations include class extension declarations, signature declarations, and method and field method declarations. The effect of a privacy annotation on these declarations is as follows:

- On a class extension declaration, a privacy annotation controls visibility of the knowledge that one class type is a subtype of another. The privacy annotation does not affect the fact that the subclass does inherit from the superclass (which affects run-time method lookup), and that the subclass type is a subtype of the superclass type in scope of the class extension declaration. Type checking ensures that the subclass is indeed a legal subtype; hiding this fact simply restricts outside clients from exploiting it. Hidden subtyping can be useful to allow a class to inherit another for implementation purposes, while acting to outside clients as if it were a separate type. **example?**
- Similarly, on a signature declaration, a privacy annotation controls visibility of the knowledge that a function supports a certain interface. Type checking will ensure that all signatures are valid; hiding a signature simply prevents some outside clients from using the signature to license a call of the function.
- A privacy annotation on a method or field method declaration is only meaningful if the declaration includes a `signature` annotation, in which case the privacy annotation is transferred to the implicit signature declaration(s). The method or field method(s) themselves

do not have visibility separate from the visibility of the function they extend; if a function is visible to a caller, then so are all methods that extend that function. If a `signature` annotation is not specified, any privacy annotation specified with a method or field method declaration must match the accessibility of the function(s) being extended, and has no other effect.

refine this semantics to say which outside clients see the subtyping & signature decls (just those that import? introduce `use` in addition to import?), and make it be the same as for regular class & fun declarations. how does this hiding interact with ITC?

note that the UW Diesel implementation doesn't yet support hiding subtyping or signature decls; all are implicitly globally visible, i.e., visible whenever their endpoint types or their function & argument types are.

explain how all declarations are visible globally for the purposes of ITC, e.g., for each function, for all its signatures (even hidden ones), enumerating all possible concrete classes (even hidden ones) that conform (even via hidden subtyping) to the signature's argument types. no modular ITC :(.

Some declarations do not allow a privacy annotation to be specified. Precedence declarations have global visibility, even when written inside a module, and so do not support a privacy annotation. **[It would be nice, and seems feasible, to make precedence declarations scoped.]** Similarly, `prim` declarations have visibility in all later `prim` declarations and in all `prim` statements occurring in the same source file, independently of what modules they are written in, and so they do not support a privacy annotation. Since they are used solely during implementation-side type checking, `disjoint`, `cover`, and `divide` declarations are globally visible and so do not admit a privacy annotation.

It is possible for a declaration to be given one degree of visibility, but to reference things that have less visibility. For example, a public class declaration might inherit from a private superclass, or a public function might have a private argument or result type. This is legal. Clients of such declarations treat such non-visible aspects of declarations as being unknown, which may prevent the client from performing some operations. For example, a client cannot exploit any subtyping from a class to a hidden superclass (the fact that every class inherits from `any` is always known), and a client cannot call a function that takes a hidden argument type, nor do anything with the result of a function returning a hidden type except treat it as being some unknown subtype of `any`. It is not even known that two occurrences of the same hidden type are the same. **the UW Diesel implementation probably does not enforce these limitations fully.**

An alternative to privacy annotations is signature ascription, as in ML. In that model, a separate "module type," called a signature in ML, is assigned to the module, and this type can hide some names by listing only a subset of the members of the module. Signature ascription does not easily let a module have multiple external views, e.g. anyone vs. submodule vs. friend, but it does allow a module's interface to be written down separately from its implementation, which can improve clarity and modular reasoning at the cost of duplicating the "headers" of externally visible members.

5.3 Qualified Names

A client can refer to a named member of a module using a qualified name, of the form *moduleName*`$`*memberName*. The member named *memberName* must be declared directly in the module named *moduleName* (not in one of its lexically enclosing scopes), and *memberName* must be accessible to the scope containing the qualified name occurrence. A module may itself be nested inside module, and a qualified name may be used to name it, i.e., *moduleName* may itself be a qualified name. Thus, a qualified name can specify a path, *outerModuleName*`$`*nestedModuleName*`$...$`*lastModuleName*`$`*memberName*, starting with some outer module (whose name is declared in a scope lexically enclosing the qualified name occurrence), then stepping through a sequence of nested modules (each of which is accessible to the scope containing the qualified name occurrence), finishing with the name of some member of the last module.

For the purposes of qualified names, the global scope is treated as a module whose name is empty. Thus, a qualified name can start with `$`, e.g., `$`*globalModuleName*`$...$`*nestedModuleName*`$`*memberName* or `$`*globalName*. Such a *fully qualified name* is a context-insensitive way to refer to a named declaration.

5.4 Import Declarations

Named members exported by modules can be accessed using qualified names, but doing so can lead to verbose programs. Alternatively, all names exported by some module can be imported into a client scope using an `import` declaration. The effect of an import declaration is to declare local aliases of all the names in the imported modules that are accessible to the importer. These aliases can then be referred to using regular, unqualified names in their scope.

An import declaration can have a privacy annotation. This annotation controls the visibility of the aliases to clients of the importing module. The aliases are known to the client of the importing module to be equivalent to the names in the importee module, in case both modules are accessible to the client.

A member declared in a scope shadows a member of the same name imported from another module. A member imported into a scope shadows a member declared in a lexically enclosing scope.

If two different modules are imported that declare the same name, and those names are not known to be aliases, then the name is considered ambiguously imported, and it cannot be referred to under its alias name. The ambiguity can be resolved using qualified names to indicate explicitly which module's version of the name is desired.

To avoid circular dependencies in name resolution, `import` declarations are ignored when resolving the names of modules in `import` declarations themselves. Only regular lexical scoping is used when resolving the names of importees.

A common idiom is to declare a nested module containing `public` and `private` members, and then immediately import the nested module into the enclosing scope. This has the effect of making

the `public` members of the nested module visible to the rest of the enclosing scope, while hiding the `private` members, analogously to how a `local` declaration in ML allows hiding of some declarations while exporting others. The imported `public` members can be made available to outside scopes by giving the `import` declaration an appropriate privacy annotation.

It is not currently possible to selectively import a subset of the names of a module, or to assign different imported names different privacies. Such an ability would be useful to add to the language, however. **how about coupling a signature ascription to an import decl? or to a module itself? seems like it could solve lots of problems....**

investigate allowing import in dyn scopes (only a limitation of the implementation, not the semantics)

5.5 Extends Declarations

A module can specify that it is a "submodule" of another module by using an `extends` declaration specifying the other module. This grants the submodule access to the `protected` members as well as the `public` members of the extended module. In addition, an `extends` declaration acts like an `import` declaration, introducing local aliases for all `public` and `protected` named members of the extended module into the submodule. The rules for privacy annotations and shadowing of these local aliases are the same as for local aliases due to `import` declarations.

An `extends` declaration is intended to achieve for modules an effect similar to what traditional class-based languages achieve for `protected` members. For example, a class and its operations can be declared in one module, with any members intended for use only by subclasses marked protected. A subclass can then be declared in a separate module, which can gain access to the `protected` members, e.g., to override them with subclass-specific methods, by declaring that the module containing the subclass extends the module containing the superclass. Note, however, that there is no obligation that the submodule actually declare a subclass, or access the `protected` members solely on behalf of subclass instances; `protected` privacy is only advisory. It would be useful to extend the language to enforce the constraint that submodules only access `protected` members of the extended module on behalf of instances of classes declared in the submodule. **introduce protected types.**

As with `import` declarations, `extend` declarations are ignored when resolving the names of modules in `import` or `extend` declarations. Only regular lexical scoping is used when resolving the names of extendees.

5.6 Friend Declarations

A module can grant an outside module access to all of its members, including its `private` members, by listing the outside module in a `friend` declaration. Then, whenever the friend module accesses a member of the module, either directly via a qualified name or indirectly through `import` or `extends` declarations, access is always granted.

In contrast to `extends` declarations, which occur in the outside extending modules that wish to gain additional access to a module's members, `friend` declarations occur in the module wishing to grant access to outside modules.

Since a `friend` declaration introduces no names, it can have no privacy annotation.

As with `import` and `extend` declarations, `friend` declarations are ignored when resolving the names of modules in `import`, `extend`, or `friend` declarations. Only regular lexical scoping is used when resolving the names of friends.

A `friend` declaration can only appear inside a module, not at top-level.

5.7 Module Extension Declarations

A module's body can be augmented "from the outside" of its original declaration using a module extension declaration, analogously to how a class or object extension declaration can augment the inheritance declarations and field initializers of a class or object "from the outside." Such a declaration begins with the keyword `extend` followed by a normal module declaration. However, unlike a regular module declaration, the module extension does not declare a new module, but instead must refer to an existing module. The body of the module extension is in the same scope as the body of the original module declaration, and the bodies of any other module extensions of the same module; all these bodies can refer to any declarations in any of the bodies. Executing a module extension causes its body statements and declarations to be executed, in the order given in the module extension; the body of the module extension is not executed when the original module is executed.

Note that a module extension declaration is unrelated to an extends declaration identifying a submodule. One begins `extend module`, while the other begins `extends`. The former augments an existing module in place, while the latter is a component of a module that claims access to `protected` members of some other modules.

Module extensions are intended to be a way for a module's implementation to be spread across multiple files. They also allow a single module to act as an open-ended "name space" container into which other files add declarations, e.g. nested modules. This ability is similar to packages in Java and namespaces in C#.

The current syntax of module extensions requires the extendee module to be referred to by a simple name, not a qualified name. To achieve the effect of an extension of a module accessed by a qualified name, e.g.

```
extend module M1$M2$M3 { ... }
```

a nested sequence of module extensions can be declared, e.g.

```
extend module M1 { extend module M2 { extend module M3 { ... } } }
```

Nonetheless, it would be useful to regularize the language to allow qualified names in this context as well. **trivial; do this.**

5.8 Function Call Overload Resolution

As described in section 5.4, if two members of different modules but with the same name are imported into the same scope, the name is ambiguous. Normally, this prevents referencing the name. However, Diesel includes support to resolve this ambiguity for the special case of function calls. In this case, the static types of the call's arguments can aid in selecting the right function being called, without requiring qualifying the function's name. To locate the function being invoked, the implementation collects all functions that have the same name, number of explicit type parameters, and number of arguments as the call, and that are declared in any lexically enclosing scope, or are declared in any module imported or extended by a lexically enclosing scope and accessible to that scope; the usual shadowing and ambiguity rules are ignored. Then the set of functions thus collected is filtered to include only those functions having visible signatures with arguments that are supertypes of the types of the actual arguments of the call. If this filtered set still has multiple callable functions, then an "ambiguous reference" error is reported. If the filtered set is empty, then a "message not understood" error is reported. Otherwise, a unique callable function has been located, and it is the one invoked by the function call.

This form of ambiguity resolution is related to the static overloading found in languages like C++, Java, and C#. One difference is that overloading based solely on static argument type is not allowed within a single scope, only when accessing like-named functions declared in different scopes but visible in a common scope.

Currently, function ambiguity resolution based on static argument types is applied to references to functions in function calls, but not to other references to functions including method and signature declarations and field initializers. It would be useful to investigate supporting ambiguity resolution for these other function references as well.

Using static argument types to resolve ambiguities in references to functions being called is convenient (and widely used in practice) for statically typed code, but it does not help dynamically typed code. Qualified names of functions must be used wherever more than one function with a matching name are visible to a call site. This unfortunately penalizes dynamically typed code relative to statically typed code.

6 Related Work

Diesel builds on Cecil.

(The following is a discussion of the related work for Cecil. This needs to be extended and revised to include the more recent work which influenced and relates to Diesel.)

Cecil builds upon much of the work done with the Self programming language [Ungar & Smith 87, Hölzle *et al.* 91a]. Self offers a simple, pure, classless object model with state accessed via message passing just like methods. Cecil extends Self with multi-methods, copy-down and initialize-only data slots, lexically-scoped local methods and fields, object extensions, static typing, and a module system. Cecil has simpler method lookup and encapsulation rules, at least when considering only the single dispatching case. Cecil's model of object creation is different than Self's. However, Cecil does not incorporate dynamic inheritance, one of the most interesting features of Self; predicate objects are Cecil's more structured but more restricted alternative to dynamic inheritance. Freeman-Benson independently developed a proposal for adding multi-methods to Self [Freeman-Benson 89].

Common Loops [Bobrow *et al.* 86] and CLOS [Bobrow *et al.* 88, Gabriel *et al.* 91] incorporate multi-methods in dynamically-typed class-based object-oriented extensions to Lisp. Method specializations (at least in CLOS) can be either on the class of the argument object or on its value. One significant difference between Cecil's design philosophy and that in CLOS and its predecessors is that Cecil's multiple inheritance and multiple dispatching rules are unordered and report any ambiguities in the source program as message errors, while in CLOS left-to-right linearization of the inheritance graph and left-to-right ordering of the argument dispatching serves to resolve all message ambiguities automatically, potentially masking real programming errors. We feel strongly that the programmer should be made aware of potential ambiguities since automatic resolution of these ambiguities can easily lead to obscure errors in programs. Cecil offers a simpler, purer object model, optional static type checking, and encapsulation. CLOS and its predecessors include extensive support for method combination rules and reflective operations [Kiczales *et al.* 91] not present in Cecil.

Dylan [Apple 92] is a new language which can be viewed as a slimmed-down CLOS, based in a Scheme-like language instead of Common Lisp. Dylan is similar to CLOS in most of the respects described above, except that Dylan always accesses state through messages. Dylan supports a form of type declarations, but these are not checked statically, cannot be parameterized, and are treated both as argument specializers and type declarations, unlike Cecil where argument specializers and argument type declarations are distinct.

Polyglot is a CLOS-like language with a static type system [Agrawal *et al.* 91]. However, the type system for Polyglot does not distinguish subtyping from code inheritance (classes are the same as types in Polyglot), does not support parameterized or parametrically polymorphic classes or methods, and does not support abstract methods or signatures. To check consistency among multi-methods within a generic function, at least the interfaces to all multi-methods of a generic function must be available at type-check-time. This requirement is similar to that of Cecil that the whole

program be available at type-check-time to guarantee that two multi-methods are not mutually ambiguous for some set of argument objects.

Kea is a higher-order polymorphic functional language supporting multi-methods [Mugridge *et al.* 91]. Like Polyglot (and most other object-oriented languages), inheritance and subtyping in Kea are unified. Kea's type checking of multi-methods is similar to Cecil's in that multi-methods must be both complete and consistent. It appears that Kea has a notion of abstract methods as well.

Leavens describes a statically-typed applicative language NOAL that supports multi-methods using run-time overloading on the declared argument types of methods [Leavens 89, Leavens & Weihl 90]. NOAL was designed primarily as a vehicle for research on formal verification of programs with subtyping using behavioral specifications, and consequently omits theoretically unnecessary features that are important for practical programming, such as inheritance of implementation, mixed static and dynamic type checking, and mutable state. Other theoretical treatments of multi-methods have been pursued by Rouaix [Rouaix 90], Ghelli [Ghelli 91], Castagna [Castagna *et al.* 92, Castagna 95], and Pierce and Turner [Pierce & Turner 92, Pierce & Turner 93].

The RPDE^3 environment supports *subdivided methods* where the value of a parameter to the method or of a global variable helps select among alternative method implementations [Harrison & Ossher 90]. However, a method can be subdivided only for particular values of a parameter or global variable, not its class; this is much like supporting only CLOS's `eql` specializers.

A number of languages, including C++ [Stroustrup 86, Ellis & Stroustrup 90], Ada [Barnes 91], and Haskell [Hudak *et al.* 90], support static overloading on function arguments, but all overloading is resolved at compile-time based on the static types of the arguments (and results, in the case of Ada) rather than on their dynamic types as would be required for true multiple dispatching.

Trellis* supports an expressive, safe static type system [Schaffert *et al.* 85, Schaffert *et al.* 86]. Cecil's parameterized type system includes features not present in Trellis, such as implicitly-bound type variables and uniform treatment of constrained type variables. Trellis restricts the inheritance hierarchy to conform to the subtype hierarchy; it only supports `isa`-style superclasses.

POOL is a statically-typed object-oriented language that distinguishes inheritance of implementation from inheritance of interface [America & van der Linden 90]. POOL generates types automatically from all class declarations (Cecil allows the programmer to restrict which objects may be used as types). Subtyping is implicit (structural) in POOL: all possible legal subtype relationships are assumed to be in force. Programmers may add explicit subtype declarations as a documentation aid and to verify their expectations. One unusual aspect of POOL is that types and classes may be annotated with *properties*, which are simple identifiers that may be used to capture distinctions in behavior that would not otherwise be expressed by a purely syntactic interface. This ameliorates some of the drawbacks of implicit subtyping.

* Formerly known as Owl and Trellis/Owl.

Emerald is another classless object-oriented language with a static type system [Black *et al.* 86, Hutchinson 87, Hutchinson *et al.* 87, Black & Hutchinson 90]. Emerald is not based on multiple dispatching and in fact does not include support for inheritance of implementation. Types in Emerald are arranged in a subtype lattice, however.

Rapide [Mitchell *et al.* 91] is an extension of Standard ML modules [Milner *et al.* 90] with subtyping and inheritance. Although Rapide does not support multi-methods and relies on implicit subtyping, many other design goals for Rapide are similar to those for Cecil.

Some more recent languages support some means for distinguishing subtyping from inheritance. These languages include Theta [Day *et al.* 95], Java [Sun 95], and Sather [Omohundro 93]. Theta additionally supports an enhanced CLU-like where-clause mechanism that provides an alternative to F-bounded polymorphism. C++'s private inheritance supports a kind of inheritance without subtyping.

Several languages support some form of mixed static and dynamic type checking. For example, CLU [Liskov *et al.* 77, Liskov *et al.* 81] allows variables to be declared to be of type `any`. Any expression may be assigned to a variable of type `any`, but any assignments of an expression of type `any` to an expression of another type must be explicitly coerced using the parameterized `force` procedure. Cedar supports a similar mechanism through its `REF ANY` type [Teitelman 84]. Modula-3 retains the `REFANY` type and includes several operations including `NARROW` and `TYPECASE` that can produce a more precisely-typed value from a `REFANY` type [Nelson 91, Harbison 92]. Cecil provides better support for exploratory programming than these other languages since there is no source code "overhead" for using dynamic typing: variable type declarations are simply omitted, and coercions between dynamically-typed expressions and statically-typed variables are implicit. On the other hand, in Cecil it sometimes can be subtle whether some expression is statically-typed or dynamically-typed.

7 Conclusion

TO BE WRITTEN

Acknowledgments

TO BE REVISED to include Diesel-related acknowledgments.

The Cecil language design and the presentation in this document have benefitted greatly from discussions with members of the Self group including David Ungar, Urs Hölzle, Bay-Wei Chang, Ole Agesen, Randy Smith, John Maloney, and Lars Bak, with members of the Kaleidoscope group including Alan Borning, Bjorn Freeman-Benson, Michael Sannella, Gus Lopez, and Denise Draper, with the Cecil group including Claudia Chiang, Jeff Dean, Charles Garrett, David Grove, Vassily Litvinov, Vitaly Shmatikov, and Stuart Williams, and others including Peter Deutsch, Eliot Moss, John Mitchell, Jens Palsberg, Doug Lea, Rick Mugridge, John Chapin, Barbara Lerner, and Christine Ahrens. Gary Leavens collaborated with the author to refine the static type system, devise the module system, and develop an efficient typechecking algorithm. Claudia Chiang implemented the first version of the Cecil interpreter, in Self. Stuart Williams augmented this interpreter with a type checker for the monomorphic subset of the Cecil type system. Jeff Dean, Greg DeFouw, Charles Garrett, David Grove, MaryAnn Joy, Vassily Litvinov, Phiem Huynh Ngoc, Vitaly Shmatikov, Ben Teitelbaum, and Tina Wong have worked on various aspects of the Vortex optimizing compiler for object-oriented languages, a.k.a. the UW Cecil implementation. A conversation with Danny Bobrow and David Ungar at OOPSLA '89 provided the original inspiration for the Cecil language design effort.

This research has been supported by a National Science Foundation Research Initiation Award (contract number CCR-9210990), a NSF Young Investigator Award (contract number CCR-945767), a University of Washington Graduate School Research Fund grant, a grant from the Office of Naval Research (contract number N00014-94-1-1136), and gifts from Sun Microsystems, IBM Canada, Xerox PARC, Edison Design Group, and Pure Software.

More information on the Diesel language and Vortex optimizing compiler projects is available at `http://www.cs.washington.edu/research/projects/cecil`. More information on related projects of the Washington Advanced Systems for Programming (WASP) group is available at `http://www.cs.washington.edu/research/progsys/wasp`.

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.

[Agrawal *et al.* 91] Rakesh Agrawal, Linda G. DeMichiel, and Bruce G. Lindsay. Static Type Checking of Multi-Methods. In *OOPSLA '91 Conference Proceedings*, pp. 113-128, Phoenix, AZ, October, 1991. Published as *SIGPLAN Notices 26(11)*, November, 1991.

[America & van der Linden 90] Pierre America and Frank van der Linden. A Parallel Object-Oriented Language with Inheritance and Subtyping. In *OOPSLA/ECOOP '90 Conference Proceedings*, pp. 161-168, Ottawa, Canada, October, 1990. Published as *SIGPLAN Notices 25(10)*, October, 1990.

[Andersen & Reenskaug 92] Egil P. Andersen and Trygve Reenskaug. System Design by Composing Structures of Interacting Objects. In *ECOOP '92 Conference Proceedings*, pp. 133-152, Utrecht, the Netherlands, June/July 1992. Published as *Lecture Notes in Computer Science 615*, Springer-Verlag, Berlin, 1992.

[Apple 92] *Dylan, an Object-Oriented Dynamic Language*. Apple Computer, April, 1992.

[Barnes 91] J. G. P. Barnes. *Programming in Ada, 3rd Edition.* Addison-Wesley, Wokingham, England, 1991.

[Black et al. 86] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object Structure in the Emerald System. In *OOPSLA '86 Conference Proceedings*, pp. 78-86, Portland, OR, September, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[Black & Hutchinson 90] Andrew P. Black and Norman C. Hutchinson. Typechecking Polymorphism in Emerald. Technical report TR 90-34, Department of Computer Science, University of Arizona, December, 1990.

[Bobrow *et al.* 86] Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. CommonLoops: Merging Lisp and Object-Oriented Programming. In *OOPSLA '86 Conference Proceedings*, pp. 17-29, Portland, OR, September, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[Bobrow *et al.* 88] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, D. A. Moon. Common Lisp Object System Specification X3J13. In *SIGPLAN Notices 23(Special Issue)*, September, 1988.

[Borning 86] A. H. Borning. Classes Versus Prototypes in Object-Oriented Languages. In *Proceedings of the 1986 Fall Joint Computer Conference*, pp. 36-40, Dallas, TX, November, 1986.

[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, January 1997.

[Bracha & Griswold 93] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a Production Environment. In *OOPSLA '93 Conference Proceedings*, pp. 215-230, Washington, D.C., September 1993. Published as *SIGPLAN Notices 28(10)*, October 1993.

[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.

[Bracha et al. 98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In *OOPSLA '98 Conference Proceedings*, Vancouver, B.C., October, 1998.

[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. 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. In *Proceedings ECOOP '98*, Brussels, Belgium, July 1998. Springer-Verlag.

[Canning *et al.* 89] Peter S. Canning, William R. Cook, Walter L. Hill, John C. Mitchell, and William Olthoff. F-Bounded Quantification for Object-Oriented Programming. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, 1989.

[Cardelli & Wegner 85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. In *Computing Surveys 17(4)*, pp. 471-522, December, 1985.

[Cardelli 93] Luca Cardelli. An implementation of Fsub. Research Report 97, Digital Equipment Corporation Systems Research Center, 1993.

[Castagna *et al.* 92] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A Calculus for Overloaded Functions with Subtyping. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pp. 182-192, San Francisco, June, 1992. Published as *Lisp Pointers 5(1)*, January-March, 1992.

[Castagna 95] Giuseppe Castagna. Covariance and Contravariance: Conflict without a Cause. In *ACM Transactions on Programming Languages and Systems 17(3)*, pp. 431-447, May 1995.

[Chambers *et al.* 89] Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of Self, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *OOPSLA '89 Conference Proceedings*, pp. 49-70, New Orleans, LA, October, 1989. Published as *SIGPLAN Notices 24(10)*, October, 1989. Also published in *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June, 1991.

[Chambers *et al.* 91] Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle. Parents are Shared Parts: Inheritance and Encapsulation in Self. In *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June, 1991.

[Chambers & Ungar 91] Craig Chambers and David Ungar. Making Pure Object-Oriented Languages Practical. In *OOPSLA '91 Conference Proceedings*, pp. 1-15, Phoenix, AZ, October, 1991. Published as *SIGPLAN Notices 26(10)*, October, 1991.

[Chambers 92a] Craig Chambers. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. Ph.D. thesis, Department of Computer Science, Stanford University, March, 1992.

[Chambers 92b] Craig Chambers. Object-Oriented Multi-Methods in Cecil. In *ECOOP '92 Conference Proceedings*, pp. 33-56, Utrecht, the Netherlands, June/July, 1992. Published as *Lecture Notes in Computer Science 615*, Springer-Verlag, Berlin, 1992.

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

[Chambers 93b] Craig Chambers. Predicate Classes. In *ECOOP '93 Conference Proceedings*, pp. 268-296, Kaiserslautern, Germany, July, 1993. Published as *Lecture Notes in Computer Science 707*, Springer-Verlag, Berlin, 1993.

[Chambers & Leavens 94] Craig Chambers and Gary T. Leavens. Typechecking and Modules for Multi-Methods. In *OOPSLA '94 Conference Proceedings*, pp. 1-15, Portland, OR, October 1994. Published as *SIGPLAN Notices 29(10)*, October 1994. An expanded and revised version to appear in *ACM Transactions on Programming Languages and Systems*.

[Chang & Ungar 90] Bay-Wei Chang and David Ungar. Experiencing Self Objects: An Object-Based Artificial Reality. Unpublished manuscript, 1990.

[Cook 89] W. R. Cook. A Proposal for Making Eiffel Type-Safe. In *ECOOP '89 Conference Proceedings*, pp. 57-70, Cambridge University Press, July, 1989.

[Cook *et al.* 90] William Cook, Walter Hill, and Peter Canning. Inheritance is not Subtyping. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, San Francisco, CA, January, 1990.

[Cook 92] William R. Cook. Interfaces and Specifications for the Smalltalk-80 Collection Classes. In In *OOPSLA '92 Conference Proceedings*, pp. 1-15, Vancouver, Canada, October, 1992. Published as *SIGPLAN Notices 27(10)*, October, 1992.

[Day *et al.* 95] Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Meyers. Subtypes vs. Where Clauses: Constraining Parametric Polymorphism. In *Proceedings of the 1995 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '95)*, pp. 156-168, Austin, TX, October 1995.

[Dean & Chambers 94] Jeffrey Dean and Craig Chambers. Towards Better Inlining Decisions Using Inlining Trials. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pp. 273-282, Orlando, FL, June 1994. Published as *Lisp Pointers 7(3)*, July-September 1994.

[Dean *et al.* 95a] Jeffrey Dean, Craig Chambers, and David Grove. Selective Specialization in Object-Oriented Languages. In *Proceedings of the 1995 SIGPLAN Conference on Programming Language Design and Implementation (PLDI '95),* La Jolla, CA, June 1995.

[Dean *et al.* 95b] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '95)*, Århus, Denmark, August 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. 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.

[Ellis & Stroustrup 90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1990.

[Freeman-Benson 89] Bjorn N. Freeman-Benson. A Proposal for Multi-Methods in Self. Unpublished manuscript, December, 1989.

[Gabriel *et al.* 91] Richard P. Gabriel, Jon L White, and Daniel G. Bobrow. CLOS: Integrating Object-Oriented and Functional Programming. In *Communications of the ACM 34(9)*, pp. 28-38, September, 1991.

[Ghelli 91] Giorgio Ghelli. A Static Type System for Message Passing. In *OOPSLA '91 Conference Proceedings*, pp. 129-145, Phoenix, AZ, October, 1991. Published as *SIGPLAN Notices 26(11)*, November, 1991.

[Goldberg & Robson 83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA, 1983.

[Goldberg 84] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, Reading, MA, 1984.

[Grove *et al.* 95] David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. Profile-Guided Receiver Class Prediction. In *Proceedings of the 1995 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '95)*, Austin, TX, October 1995.

[Grove 95] David Grove. The Impact of Interprocedural Class Analysis on Optimization. In *Proceedings of CASCON '95*, pp. 195-203, Toronto, Canada, November 1995.

[Halbert & O'Brien 86] Daniel C. Halbert and Patrick D. O'Brien. Using Types and Inheritance in Object-Oriented Languages. Technical report DEC-TR-437, Digital Equipment Corp., April, 1986.

[Harbison 92] Samuel P. Harbison. *Modula-3*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[Harrison & Ossher 90] William Harrison and Harold Ossher. Subdivided Procedures: A Language Extension Supporting Extensible Programming. In *Proceedings of the 1990 International Conference on Computer Languages*, pp. 190-197, New Orleans, LA, March, 1990.

[Harrison & Ossher 93] William Harrison and Harold Ossher. Subject-Oriented Programming (A Critique of Pure Objects). In *OOPSLA '93 Conference Proceedings*, pp. 411-428, Washington, D.C., September 1993. Published as *SIGPLAN Notices 28(10)*, October 1993.

[Hölzle *et al.* 91a] Urs Hölzle, Bay-Wei Chang, Craig Chambers, Ole Agesen, and David Ungar. *The Self Manual, Version 1.1*. Unpublished manual, February, 1991.

[Hölzle *et al.* 91b] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Programming Languages with Polymorphic Inline Caches. In *ECOOP '91 Conference Proceedings*, pp. 21-38, Geneva, Switzerland, July, 1991.

[Hölzle *et al.* 92] Urs Hölzle, Craig Chambers, and David Ungar. Debugging Optimized Code with Dynamic Deoptimization. To appear in *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, CA, June, 1992.

[Hölzle 93] Urs Hölzle. Integrating Independently-Developed Components in Object-Oriented Languages. In *ECOOP '93 Conference Proceedings*, pp. 36-56, Kaiserslautern, Germany, July 1993. Published as *Lecture Notes in Computer Science 707*, Springer-Verlag, Berlin, 1993.

[Hudak *et al.* 90] Paul Hudak, Philip Wadler, Arvind, Brian Boutel, Jon Fairbairn, Joseph Fasel, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Simon Peyton Jones, Mike Reeve, David Wise, Jonathan Young. *Report on the Programming Language Haskell, Version 1.0*. Unpublished manual, April, 1990.

[Hutchinson 87] Norman C. Hutchinson. *Emerald: An Object-Based Language for Distributed Programming.* Ph.D. thesis, University of Washington, January, 1987.

[Hutchinson *et al.* 87] Norman C. Hutchinson, Rajendra K. Raj, Andrew P. Black, Henry M. Levy, and Eric Jul. The Emerald Programming Language Report. Technical Report 87-10-07, Department of Computer Science, University of Washington, October, 1987.

[Ingalls 86] Daniel H. H. Ingalls. A Simple Technique for Handling Multiple Polymorphism. In *OOPSLA '86 Conference Proceedings*, pp. 347-349, Portland, OR, September, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[Jenks & Sutor 92] Richard D. Jenks and Robert S. Sutor. *Axiom: the Scientific Computing System*. Springer-Verlag. 1992.

[Kiczales *et al.* 91] Gregor Kiczales, James des Rivières, and Daniel G. Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, Cambridge, MA, 1991.

[Kristensen *et al.* 87] B. B. Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. The BETA Programming Language. In *Research Directions in Object-Oriented Programming*, MIT Press, Cambridge, MA, 1987.

[LaLonde *et al.* 86] Wilf R. LaLonde, Dave A. Thomas, and John R. Pugh. An Exemplar Based Smalltalk. In *OOPSLA '86 Conference Proceedings,* pp. 322-330, Portland, OR, September, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[Leavens 89] Gary Todd Leavens. *Verifying Object-Oriented Programs that use Subtypes*. Ph.D. thesis, MIT, 1989.

[Leavens & Weihl 90] Gary T. Leavens and William E. Weihl. Reasoning about Object-Oriented Programs that use Subtypes. In *OOPSLA/ECOOP '90 Conference Proceedings*, pp. 212-223, Ottawa, Canada, October, 1990. Published as *SIGPLAN Notices 25(10)*, October, 1990.

[Lieberman 86] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *OOPSLA '86 Conference Proceedings*, pp. 214-223, Portland, OR, September, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[Lieberman *et al.* 87] Henry Lieberman, Lynn Andrea Stein, and David Ungar. The Treaty of Orlando. In *Addendum to the OOPSLA '87 Conference Proceedings*, pp. 43-44, Orlando, FL, October, 1987. Published as *SIGPLAN Notices 23(5)*, May, 1988.

[Liskov *et al.* 77] Barbara Liskov, Alan Snyder, Russell Atkinson, and J. Craig Schaffert. Abstraction Mechanisms in CLU. In *Communications of the ACM 20(8)*, pp. 564-576, August, 1977.

[Liskov *et al.* 81] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*. Springer-Verlag, Berlin, 1981.

[Liskov et al. 94] Barbara Liskov, Dorothy 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.

[Litvinov 98] Vassily Litvinov. Constraint-Based Polymorphism in Cecil:Towards a Practical and Static Type System. In *OOPSLA '98 Conference Proceedings*, Vancouver, B.C., October, 1998.

[Madsen & Møller-Pedersen 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, Reading, MA, 1993.

[Meyer 86] Bertrand Meyer. Genericity versus Inheritance. In *OOPSLA '86 Conference Proceedings*, pp. 391-405, Portland, OR, September, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[Meyer 88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, New York, 1988.

[Meyer 92] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, New York, 1992.

[Milner *et al.* 90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[Mitchell *et al.* 91] John Mitchell, Sigurd Meldal, and Neel Hadhav. An Extension of Standard ML Modules with Subtyping and Inheritance. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, Williamsburg, VA, January, 1991.

[Moon 86] David A. Moon. Object-Oriented Programming with Flavors. In *OOPSLA '86 Conference Proceedings,* pp. 1-8, Portland, OR, September, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[Mugridge *et al.* 91] W. B. Mugridge, J. G. Hosking, and J. Hamer. Multi-Methods in a Statically-Typed Programming Language. Technical report #50, Department of Computer Science, University of Auckland, 1991. Also in *ECOOP '91 Conference Proceedings*, Geneva, Switzerland, July, 1991.

[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, January 1997.

[Nelson 91] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[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, January 1997.

[Omohundro 93] Stephen Omohundro. *The Sather 1.0 Specification*. Unpublished manual, June 1993.

[Pierce & Turner 92] Benjamin C. Pierce and David N. Turner. Statically Typed Multi-Methods via Partially Abstract Types. Unpublished manuscript, October, 1992.

[Pierce & Turner 93] Benjamin C. Pierce and David N. Turner. Object-Oriented Programming Without Recursive Types. In *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages*, January, 1993.

[Pierce & Turner 98] Benjamin C. Pierce and David N. Turner. Local Type Inference. In *Conference Record of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 252–265, January 1998.

[Rees & Clinger 86] Jonathan Rees and William Clinger, editors. *Revised^3 Report on the Algorithmic Language Scheme*. In *SIGPLAN Notices 21(12)*, December, 1986.

[Rouaix 90] Francois Rouaix. Safe Run-Time Overloading. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, pp. 355-366, San Francisco, CA, January, 1990.

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

[Santas 93] Philip S. Santas. A Type System for Computer Algebra. In *International Symposium on Symbolic and Algebraic Computation*. 1993.

[Schaffert *et al.* 85] Craig Schaffert, Topher Cooper, and Carrie Wilpolt. Trellis Object-Based Environment, Language Reference Manual. Technical report DEC-TR-372, November, 1985.

[Schaffert *et al.* 86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An Introduction to Trellis/Owl. In *OOPSLA '86 Conference Proceedings*, pp. 9-16, Portland, OR, September, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[Snyder 86] Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *OOPSLA '86 Conference Proceedings*, pp. 38-45, Portland, OR, September, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[Steele 84] Guy L. Steele Jr. *Common LISP.* Digital Press, 1984.

[Stroustrup 86] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 1986.

[Sun 95] Sun Microsystems. *The Java Language Specification.* Unpublished manual, May 1995.

[Teitelman 84] Warren Teitelman. *The Cedar Programming Environment: A Midterm Report and Examination*. Xerox PARC technical report CSL-83-11, June, 1984.

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

[Touretzky 86] D. Touretzky. *The Mathematics of Inheritance Systems.* Morgan-Kaufmann, 1986.

[Ungar & Smith 87] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *OOPSLA '87 Conference Proceedings*, pp. 227-241, Orlando, FL, October, 1987. Published as *SIGPLAN Notices 22(12)*, December, 1987. Also published in *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June, 1991.

[Ungar *et al.* 91] David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle. Organizing Programs without Classes. In *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June, 1991.

[Ungar 95] David Ungar. Annotating Objects for Transport to Other Worlds. In *Proceedings of the 1995 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '95)*, pp. 73-87, Austin, TX, October 1995.

[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.

[Watt *et al.* 88] Steven M. Watt, Richard D. Jenks, Robert S. Sutor, and Barry M. Trager. The Scratchpad II Type System: Domains and Subdomains. In *Proceedings of the International Workshop on Scientific Computation*, Capri, Italy, 1988. Published in *Computing Tools for Scientific Problem Solving*, A. M. Miola, ed., Academic Press, 1990.

Appendix A Annotated Diesel Syntax

In our EBNF notation, vertical bars (|) are used to separate alternatives. Braces ({ . . . }) surround strings that can be repeated zero or more times. Brackets ([. . .]) surround an optional string. Parentheses are used for grouping. Literal tokens are included in quotation marks (**" . . . "**).

A.1 Grammar

a program is defined by a (root) file, which contains a sequence of declaration blocks and statements

```
program         ::= file_body
file_body       ::= { top_decl | stmt }
```

declarations that can appear at top level:

```
top_decl        ::= include_decl
                |   static_decl
                |   dyn_decl
```

declarations that can appear immediately within a module:

```
mod_decl        ::= friend_decl
                |   static_decl
                |   dyn_decl
```

declarations that can appear either at top level or immediately within a module, but not in the body of a function or method, i.e., any "static" context:

```
static_decl     ::= module_decl
                |   ext_module_decl
                |   import_decl
                |   extends_decl
                |   class_decl
                |   ext_class_decl
                |   predicate_decl
                |   disjoint_decl
                |   cover_decl
                |   divide_decl
                |   synonym_decl
                |   fun_decl
                |   method_decl
                |   signature_decl
                |   field_decl
                |   field_method_decl
                |   precedence_decl
                |   prim_decl
                |   pragma
```

declarations that can appear anywhere, including in the body of a function, method, or other "dynamically executed" context:

```
dyn_decl        ::= let_decl
```

include declarations specify other files, whose syntax is file_body, that should also be included in the program

```
include_decl    ::= "include" file_name {pragma} ";"
file_name       ::= string
```

modules introduce a nested namespace; the visibility of the members of a module are controlled by privacy annotations

```
module_decl     ::= [privacy] "module" name {pragma} module_body
```

```
module_body     ::= "{" module_contents "}" ["; "]
                |   "; " module_contents "end" "module" [name] "; "
                |   "; " module_contents <EOF>     module body ends at end of file
module_contents::= { mod_decl | stmt }
```

a module can be extended with additional members

```
ext_module_decl::= "extend" module_decl
```

import declarations provide local aliases, with the given visibility, for public members of the imported modules, plus protected and private members of imported modules that list the importer as a friend

```
import_decl     ::= [privacy] "import" module_refs {pragma} ";"
module_refs     ::= module_ref {"," module_ref}
```

extends declarations provide local aliases, with the given visibility, for public and protected members of the extended modules

```
extends_decl    ::= [privacy] "extends" module_refs {pragma} ";"
```

friend declarations grant access to protected and private members to the named modules

```
friend_decl     ::= "friend" module_refs {pragma} ";"
```

name a module

```
module_ref      ::= qualified_name
```

name something perhaps in another module or at the top level

```
qualified_name ::= [[module_ref] "$"] name
qualified_op_name::=[[module_ref] "$"] op_name
```

class and object declarations

```
class_decl      ::= [type_cxt] [privacy] class_kind name [formal_params]
                        [type_cons] ["isa" class_ref_ps] [field_inits]
                        {pragma} ";"
class_kind      ::= "abstract" "class"
                |   "class"
                |   "object"
                |   "prim" "class"
class_ref_ps    ::= class_ref_p { "," class_ref_p }
class_ref_p     ::= qualified_name [param_patterns]
field_inits     ::= "{" field_init { "," field_init } "}"
field_init      ::= name_fun_ref ["@" class_ref] ":=" expr
```

name a class or object

```
class_ref       ::= qualified_name [params]
```

declared classes and objects can be extended with new superclasses and field initializers

```
ext_class_decl ::= [type_cxt] [privacy] "extend" ext_class_kind class_ref_p
                        [type_cons] ["isa" class_ref_ps] [field_inits]
                        {pragma} ";"
ext_class_kind ::= "class"
                |   "object"
```

predicate class declarations declare "virtual" subclasses into which objects are dynamically classified whenever some predicate over them is true

```
predicate_decl ::= [type_cxt] [privacy] "predicate" name [formal_params]
                       [type_cons] ["isa" class_ref_ps]
                       ["when" expr] {pragma} ";"
```

this syntax below needs to be extended to refer to parameterized predicate classes, e.g. with class_ref_ps in places. also, what about privacy?

```
disjoint_decl  ::= "disjoint" class_refs ";"
cover_decl     ::= "cover" class_ref "by" class_refs ";"
divide_decl    ::= "divide" class_ref "into" class_refs ";"
class_refs     ::= class_ref { "," class_ref }
```

synonym declarations define an alternate name for an existing type

```
synonym_decl   ::= [type_cxt] [privacy] "synonym" name [formal_params]
                       "=" type_pattern [type_cons] {pragma} ";"
```

fun declarations declare new named (generic) functions, optionally with a default implementation

```
fun_decl       ::= [type_cxt] [privacy] "fun" fun_name [formal_params]
                       "(" [fun_formals] ")" [type_decl_p] [type_cons]
                       {pragma} fun_body
fun_name       ::= name | op_name
fun_formals    ::= fun_formal { "," fun_formal }
fun_formal     ::= [name] ":" type_pattern
                 |   name
fun_body       ::= method_body | ";"
```

name a function

```
fun_ref        ::= name_fun_ref | op_fun_ref
name_fun_ref   ::= qualified_name [params]
op_fun_ref     ::= qualified_op_name [params]
```

a method declaration adds a new dynamically dispatched subcase to an existing generic function; if the "signature" keyword is present, the method declaration also extends the type of the generic function with a new argument type-result type relation

```
method_decl    ::= [type_cxt] [privacy] "method" ["signature"] formal_fun_ref
                       "(" [meth_formals] ")" [type_decl_p] [type_cons]
                       {pragma} method_body
formal_fun_ref ::= name_formal_fun_ref | op_formal_fun_ref
name_formal_fun_ref::=qualified_name [formal_params]
op_formal_fun_ref::=qualified_op_name [formal_params]
meth_formals   ::= meth_formal { "," meth_formal }
meth_formal    ::= [name] ":" type_pattern
                 |   [name] "@" [["`"] name "<="] class_ref_p
                 |   name
method_body    ::= "{" (body | prim_body) "}" [";"]
```

signature declarations extend the types of existing generic functions with new argument type-result type relations

```
signature_decl ::= [type_cxt] [privacy] "signature" formal_fun_ref
                       "(" [fun_formals] ")" [type_decl_p] [type_cons]
                       {pragma} ";"
```

a field declaration (a) allocates either a storage table to hold the field contents for each possible argument object, or a single memory location if the "shared" keyword is present, (b) declares a new named (generic) function with a default implementation that is a "get accessor" for the field, optionally with code to compute the field's initial value, and (c) if the "var" keyword is present, declares a new named (generic) function with a default implementation that is a "set accessor" for the field, whose name is the name of the get accessor prefixed with "set_".

```
field_decl      ::= [type_cxt] [field_privacy] ["shared"] ["var"] "field"
                        name [formal_params]
                        "(" fun_formal ")" [type_decl_p] [type_cons]
                        {pragma} field_body
field_privacy   ::= privacy ["get" [privacy "put"] | "put"]
field_body      ::= "{" body "}" [";"] | ";"
```

a field method declaration (a) allocates either a storage table to hold the field contents for each possible argument object, or a single memory location if the "shared" keyword is present, (b) adds a new dynamically dispatched "get accessor" subcase to an existing generic function, and (c) if the "var" keyword is present, adds a new dynamically dispatched "set accessor" subcase to an existing generic function, whose name is the name of the get accessor prefixed with "set_". if the "signature" keyword is present, the field method declaration also extends the type(s) of the generic function(s) with new argument type-result type relations

```
field_method_decl::=[type_cxt] [field_privacy] ["shared"] ["var"] "field"
                        "method" ["signature"] name_formal_fun_ref
                        "(" meth_formal ")" [type_decl_p] [type_cons]
                        {pragma} field_body
```

precedence declarations control the precedence and associativity of binary operators

```
precedence_decl::= "precedence" op_names [associativity] {precedence}
                        {pragma} ";"
associativity   ::= "left_associative" | "right_associative" | "non_associative"
precedence      ::= "below" op_names | "above" op_names | "with" op_names
op_names        ::= op_name { "," op_name }
```

primitive declarations include an arbitrary piece of code in the compiled file (implementation specific)

```
prim_decl       ::= prim_body ";"
```

primitive bodies support access to code written in other languages (implementation specific)

```
prim_body       ::= "prim" { language_binding }
language_binding::= language ":" code_string
                |   language "{" code_chars "}"
language        ::= name
code_string     ::= string
code_chars      ::= brace_balanced_chars      any characters, with balanced use of "{" and "}"
```

variable declarations bind names to objects; if "var" is present then the variable is assignable

```
let_decl        ::= [privacy] "let" ["var"] name [type_decl] {pragma}
                        ":=" expr ";"
```

name a variable

```
var_ref         ::= qualified_name
```

privacy of a declaration; defaults to protected. a privacy annotation is meaningless for top-level declarations

```
privacy         ::= "public" | "protected" | "private"
```

body of a function, method, field initializer, closure, or parenthetical subexpression

```
body           ::= {dyn_decl | stmt} result
                 |   empty                          return void
stmt           ::= assignment ";"
                 |   expr ";"
result         ::= normal_return                    return an expression
                 |   non_local_rtn                  return from the lexically-enclosing method
normal_return  ::= dyn_decl                         return void
                 |   assignment [";"]               return void
                 |   expr [";"]                     return result of expression
non_local_rtn  ::= "^" [";"]                        do a non-local return, returning void
                 |   "^" expr [";"]                 do a non-local return, returning a result
```

assignment-like statement

```
assignment     ::= var_ref ":=" expr                assign to a mutable variable; returns void
                 |   assign_msg                     assignment-like syntax for messages
assign_msg     ::= lvalue_msg ":=" expr             sugar for set_lmsg (lexprs...,rexpr)
lvalue_msg     ::= message
                 |   dot_msg
                 |   unop_msg
                 |   binop_msg
```

expressions

```
expr           ::= binop_expr
```

binary msgs have lowest precedence

```
binop_expr     ::= binop_msg | unop_expr
binop_msg      ::= binop_expr op_fun_ref binop_expr
                                                    predecence and associativity as declared
```

unary msgs have second-lowest precedence

```
unop_expr      ::= unop_msg | dot_expr
unop_msg       ::= op_fun_ref unop_expr             & and ^ are not allowed as unary operators
```

dotted messages have second-highest precedence

```
dot_expr       ::= dot_msg | simple_expr
dot_msg        ::= dot_expr "." name_fun_ref ["(" [exprs] ")"]
                                  sugar for name_fun_ref(dot_expr,exprs...)
```

simple messages have highest precedence

```
simple_expr    ::= literal
                 |   var_expr
                 |   vector_expr
                 |   closure_expr
                 |   object_expr
                 |   message
                 |   resend
                 |   paren_expr
```

literal constants

```
literal        ::= integer
                 |   single_float
```

```
                |    double_float
                |    character
                |    string
```

reference a variable or a named object

```
var_expr        ::= var_ref                     reference a variable
                |   class_ref                   reference an object declaration
```

build a vector, optionally specifying the element type

```
vector_expr     ::= "[" [":" type ":"] [exprs] "]"
exprs           ::= expr { "," expr }
```

build a closure

```
closure_expr    ::= ["&" "(" [closure_formals] ")" [type_decl]] "{" body "}"
closure_formals::= closure_formal { "," closure_formal }
closure_formal  ::= [name] ":" type
                |   name
```

build a new instance of a concrete class or object

```
object_expr     ::= "new" class_ref [field_inits]
```

send a message

```
message         ::= name_fun_ref "(" [exprs] ")"
```

resend the message to an overridden method case

```
resend          ::= "resend" ["(" resend_args ")"]
resend_args     ::= resend_arg { "," resend_arg }
resend_arg      ::= expr                        corresponding formal of sender must be
                                                  unspecialized
                |   name                        undirected resend (name is a specialized formal)
                |   name "@" class_ref          directed resend (name is a specialized formal)
```

introduce a new nested scope

```
paren_expr      ::= "(" body ")"
```

a type declaration is an annotation giving an explicit type to a variable or result

```
type_decl       ::= ":" type
type_decl_p     ::= ":" type_pattern
```

type contexts and constraints

```
type_cxt        ::= "forall" formal_param { "," formal_param } [type_cons] ":"
                |   "forall" type_cons ":"
type_cons       ::= "where" type_constraint { "," type_constraint }
type_constraint::= sub_constraint | sig_constraint | type_pattern
sub_constraint  ::= type_pattern ("<=" | ">=") type_pattern
sig_constraint  ::= ["signature"] formal_fun_ref
                        "(" [fun_formals] ")" type_decl_p
```

syntax of types

```
types           ::= type { "," type }
type            ::= lub_type
```

```
lub_type        ::= lub_type "|" glb_type
                |   glb_type
glb_type        ::= glb_type "&" simple_type
                |   simple_type
simple_type     ::= named_type
                |   closure_type
                |   "(" type ")"
named_type      ::= class_ref                   a class, object, or synonym
closure_type    ::= "&" "(" [arg_types] ")" [type_decl]
arg_types       ::= arg_type { "," arg_type }
arg_type        ::= [[name] ":"] type
```

type patterns are types that can contain binding occurrences of implicit type parameters

```
type_patterns   ::= type_pattern { "," type_pattern }
type_pattern    ::= lub_type_p
lub_type_p      ::= lub_type_p "|" glb_type_p
                |   glb_type_p
glb_type_p      ::= glb_type_p "&" simple_type_p
                |   simple_type_p
simple_type_p   ::= binding_type_p
                |   named_type_p
                |   closure_type_p
                |   "(" type_pattern ")"
binding_type_p  ::= "`" name_binding
named_type_p    ::= class_ref_p                 a class, object, or synonym
closure_type_p  ::= "&" "(" [arg_type_ps] ")" [type_decl_p]
arg_type_ps     ::= arg_type_p { "," arg_type_p }
arg_type_p      ::= [[name] ":"] type_pattern
```

`name_binding` introduces a type variable called `name`

```
name_binding    ::= name [">=" type_pattern] ["<=" type_pattern]
```

formal type parameters for objects and methods

```
formal_params   ::= "[" formal_param { "," formal_param } "]"
formal_param    ::= ["`"] name_binding
```

actual type parameters for objects and methods

```
params          ::= "[" types "]"
```

actual type parameters for types that may contain binding occurrences of implicit type variables

```
param_patterns  ::= "[" type_patterns "]"
```

pragmas can be added at various points in a program to provide implementation-specific hints/commands

```
pragma          ::= "(**" exprs "**)"
```

A.2 Tokens

Bold-faced non-terminals in this grammar are the tokens in the full grammar of A.1. As usual, tokens are defined as the longest possible sequence of characters that are in the language defined by the grammar given below. The meta-notations "`one of "..."`", "`any but` *x*," and "*x*`..`*y*"

are used to concisely list a range of alternative characters. `space`, `tab`, and `newline` stand for the corresponding characters.

```
name            ::= letter {letter | digit} [id_cont]
                |   "_" {"_"} op_name          the first underscore is not part of the name
op_name         ::= punct {punct} [id_cont]
                |   "_" {"_"} name             the first underscore is not part of the name
id_cont         ::= "_" {"_"} [name | op_name]

integer         ::= [radix] hex_digits         a leading "-" is considered a unary operator
radix           ::= digits "_"
hex_digits      ::= hex_digit {hex_digit}
hex_digit       ::= digit | one of "a..fA..F"

single_float    ::= float
double_float    ::= float ("d"|"D")
float           ::= digits "." digits [exponent]
                |   digits exponent
exponent        ::= ("e"|"E") ["+" | "-"] digits
digits          ::= digit {digit}

character       ::= "'" char "'"
string          ::= """ { char | line_break } """
char            ::= any | "\" escape_char
escape_char     ::= one of "abfnrtvABFNRTV'"\?"
                |   ["d"|"D"] digit [digit [digit]]  decimal character code
                |   ("o"|"O") digit [digit [digit]]  octal character code
                |   ("x"|"X") hex_digit [hex_digit]  hexadecimal character code
line_break      ::= "\" {whitespace} new_line {whitespace} "\"
                                               characters between and including \'s
                                               are not part of the string

brace_balanced_chars ::=
                    {any but "{"} ["{" brace_balanced_chars "}" {any but "}"}]

letter          ::= one of "a..zA..Z"
digit           ::= one of "0..9"
punct           ::= one of "!#%^&*-+=<>/?~\|"
```

A.3 White Space

Whitespace is allowed between any pair of tokens in the grammar in A.1.

```
whitespace      ::= space | tab | newline | comment
comment         ::= "--" {any but newline} newline   comment to end of line
                |   "(--" {any} "--)"          bracketed comment; can be nested
```

Index

NOTE: this index is in the process of being populated

A

abstract class 8

C

class
    declaration of 7
concrete class 8

I

include declaration 6

L

let declaration 7

N

named object declaration
    see object declaration

O

object declaration 8

V

var annotation
    on fields 19
    on variables 7