Contextual Polymorphism

by

Glen Jeffrey Ditchfield

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 1994

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Programming languages often provide for various sorts of static type checking as a way of detecting invalid programs. However, statically checked type systems often require unnecessarily specific type information, which leads to frustratingly inflexible languages. Polymorphic type systems restore flexibility by allowing entities to take on more than one type.

This thesis discusses polymorphism in statically typed programming languages. It provides precise definitions of the term "polymorphism" and for its varieties, "*ad-hoc* polymorphism", "universal polymorphism", "inclusion polymorphism", and "parametric polymorphism", and surveys and compares many existing mechanisms for providing polymorphism in programming languages.

Finally, it introduces a new polymorphism mechanism, *contextual polymorphism*. Contextual polymorphism is a variant of parametric polymorphism that is based on *contexts*, which are abstractions of collections of declarations, and *assertions*, which link polymorphic routines to the environments that call them. Contexts themselves provide a useful structuring mechanism for software, because they represent the notions (and relationships between notions) that programmers have in mind when they design programs. Contextual polymorphism avoids many problems associated with other polymorphism mechanisms, while preserving their benefits. The formal definition of these language constructs is given in terms of an extension of F_ω, the ω-order polymorphic typed lambda calculus. The practicality of the constructs is shown by a discussion of Cforall, an extension of the programming language C that supports contextual polymorphism.

Acknowledgements

I would like to thank Professor Don Cowan, Professor Doug Lea, and Professor Bruno Preiss, for agreeing to serve on my examining committee and for their many helpful suggestions and questions; Professor Gord Cormack, Professor Dominic Duggan, Dr. John Ophel, and Dr. Bob Zarnke for comments and conversations stretching back over many years; and my supervisor, Professor Peter Buhr, for not letting me quit.

Dedication

To Helen Cameron, for the past and the future.

Contents

Chapter 1

Introduction

Expressiveness and safety are two important criteria for judging programming languages. Informally, the expressiveness of a language measures the variety of useful programs that can be written with it, and its safety measures the variety of meaningless programs that can *not* be written with it. These attributes can conflict with each other to some extent, and part of the art of programming language design lies in balancing their demands.

Many languages provide strong type checking as a safety feature. Every data item has an associated type, which defines the set of operations that can be applied to the data; if a program attempts to apply an operation to data of the wrong type, the language's translator or run-time system detects and reports the error (as opposed to silently allowing the program to attempt a meaningless computation). Type checking can occur statically or dynamically. *Static checking* occurs before program execution, based on information about the types of values deduced from the text of the program. In some programming languages, the actual type of a value during execution may differ from its statically determined type. In those cases, a static type checker must make conservative assumptions. *Dynamic checking* occurs during execution, and can use exact type information. For some programs a programmer may be able to prove that a program will encounter no type errors when it executes, even though that "theorem" is beyond the capabilities of a static type checker for that programming language. In such cases, dynamic type checking provides greater expressiveness. However, programmers rarely attempt proofs of dynamic type-safety because of their difficult nature, and programming errors are notoriously plentiful, so static type checking provides greater safety.

This conflict has led to a search for programming languages that are statically checkable but

less restrictive. Ideally, no useful, semantically well-defined computation would be statically type-unsafe. Polymorphic languages aim for that goal by allowing programmers to define computations that are valid for operands of many different types.

This thesis examines polymorphic languages in detail. The remainder of this chapter defines some useful terms and gives some criteria for comparing languages. Chapter two reviews the kinds of polymorphism, and surveys a number of existing polymorphism mechanisms. Chapter three discusses a novel polymorphism mechanism, called contextual polymorphism. Chapter four demonstrates contextual polymorphism in the context of Cforall, an extension of the C programming language.

1.1 Definitions and Notations

A set of values that can be represented by a computer and that is distinguishable from all other such sets is a *type*. Different languages use different rules to distinguish between sets. One programming language might allow a program to contain two types that implement the notion of "complex number", each represented by a pair of floating-point numbers, but with one type interpreted as rectangular coordinates and the other as polar coordinates. A different language might treat all such pairs as having the same type.

A *type generator* defines types, based on other types and values; for example, many languages have an array type generator that defines array types based on an element type and bounding values of an index type.

A *notion* is a concept, such as "cosine" or "dictionary" or "file merge" or "six". An *implementation* is a programming-language text that defines computations that allow a computer to provide the behaviour associated with a notion. An *interface* is a programming-language text that provides information about the use and behaviour of implementations. Interfaces differ from types in that they may include extra information such as routine pre- and post-conditions; also, some languages provide entities such as modules, which have interfaces but are not values and hence do not have types.

An *operation* is a notion such as "add" or "sort", which defines the effect of a calculation. An *algorithm* is a notion that describes a sequence of calculations that perform an operation. A *routine* is an implementation of an algorithm; this includes pre-defined routines and operators like "+" and `abs`.

A group of programming-language statements that use an implementation is called a *client* of the implementation.

Discussions that do not depend on a specific programming language will use the following notation.

- e and e_i denote arbitrary expressions.
- σ, σ_i, τ and τ_i denote arbitrary types.
- $e : \tau$ means "the value of e has type τ".
- The type $\tau_1 \rightarrow \tau_2 \rightarrow \cdots \tau_n$ is the type of a routine taking an arguments of type $\tau_1, \tau_2, \ldots$ and returning a value of type τ_n. The sine function has type real→real, and the real addition function has type real→real→real.
- The expression $\lambda x_1 : \tau_1 \cdot \lambda x_2 : \tau_2 \cdot \ldots \cdot e$, where e is an expression of type τ_n, is a routine with type $\tau_1 \rightarrow \tau_2 \rightarrow \cdots \tau_n$ and with parameters named $x_1, x_2, \ldots$.
- Record types use the notation "$\langle \cdots \times \cdots \rangle$", so $\langle f_1 : \tau_1 \times f_2 : \tau_2 \rangle$ is a record type with two *fields* named f_1 and f_2, with types τ_1 and τ_2, respectively.
- The expression "$\langle f_1 := e_1, f_2 := e_2 \rangle$" creates a record value.
- Inference rules have the form

$$\frac{p \qquad q \qquad \ldots \qquad r}{s}$$

  meaning that if p, q, ..., and r can be proven, then s can be proven. For example, the usual type-checking rule for routine calls is

$$\frac{e_1 : \tau_1 \rightarrow \tau_2 \qquad e_2 : \tau_1}{e_1(e_2) : \tau_2}$$

  or, in English, "if e_1 is a routine taking an argument of type τ_1 and returning a value of type τ_2, and if e_2 has type τ_1, then applying e_1 to e_2 results in a value of type τ_2". The rule for record field extraction is

$$\frac{e : \langle f_1 : \tau_1 \times f_2 : \tau_2 \times \ldots \times f_n : \tau_n \rangle}{e.f_i : \tau_i}$$

The definitions of many other terms and notations appear at their first use. The index contains an entry for every definition.

1.2 Desirable Properties

Many authors have listed many properties that they believed were characteristics of good programming languages. Of those properties, the following seem particularly relevant to issues that arise when comparing polymorphic programming languages.

Efficiency It should be possible to implement the features of a language efficiently enough so that programmers rarely feel the need to avoid them or work around them in the interests of conserving computer resources.

> "If anyone is to be allowed to introduce inefficiency, it should be the user programmer, not the language designer. The user programmer can take advantage of this freedom to write better structured and clearer programs, and should not have to expend extra effort to obscure the structure and write less clear programs just to regain the efficiency which has been so arrogantly pre-empted by the language designer." [28]

Strong Static Interface Checking

> "The [programming language] should be designed to reduce as far as possible the scope for coding error; or at least to guarantee that such errors can be detected by a compiler, before the program even begins to run. Certain programming errors cannot always be detected in this way, and must be cheaply detectable at run time; in no case can they be allowed to give rise to machine or implementation dependent effects, which are inexplicable in terms of the language itself." [28]

"Strong interface checking" extends the familiar concept of strong type checking: no operation can be performed on data unless the behaviour of the operation is defined for that data. The difference between strong type checking and strong interface checking is that checking is based on the interfaces of data and operations. An interface might not provide complete type information: for instance, many languages let programmers declare "incomplete" record types, which do not specify the record's fields, and records of these types can be operated on in certain restricted ways. An interface might also describe properties that are not part of a language's type system, such as a routine's preconditions.

Static checking (where possible) is preferable to dynamic checking because it guarantees that certain classes of errors are not present in the program. If dynamic checking is used, that guarantee

can only be provided by exhaustive testing, which is at best slow and at worst impossible, or by human verification, which is expensive and error-prone. All other things being equal, the more static checks a programming language provides for, the better it is, since a larger class of errors is excluded. However, a language design must balance safety against efficiency, because some conceivable checks are infeasible or impossible: for instance, static verification of routine preconditions would in general require a full theorem prover.

Expressiveness A programming language is *expressive* if it can be used in a clear, uncontorted way to implement a wide variety of notions. One language is more expressive than another if it can express all of the notions that the other can, and other notions as well.

One aspect of expressiveness is *flexibility*: the ability to express notions that are quite different from those foreseen by the language's designers. A second is support for *polymorphic data*. The existence of polymorphic data follows from the principle of declaration correspondence, proposed by Landin [31] and discussed extensively by Harland [26], who states it this way: "*all* options or properties associated with declarations [should] be uniformly available." Hence, if programmers can define routines with parameters that can be bound to arguments with different types, it should be possible for them to define data structures with fields that can be bound to values of different types. For example, a program might contain a list of various geometric figures, such as circles and polygons; every element could have a different type. The program could safely apply any operation that is valid for any type of figure to any element of the list.

A third aspect of the expressiveness of a programming language is its ability to describe properties of notions. Notions have various properties, such as relationships between the types of the operands of an operation; different notions have different properties. Consider the notion $\min(a, b)$, which returns the smaller of two arguments. That notion has at least two important properties.

- It is associated with the notion of "ordered values", which defines the types of its operands and its returned value. An implementation of min should accept values from any of (and only) those "ordered types" that implement "ordered values".
- The arguments of min do not *just* have ordered types; they must have the *same* type. If the arguments belong to different ordered types, there may not be an ordering defined between them, even though each has its own defined ordering. Similarly, the value returned by min is not just any ordered value; it has exactly the same type as the argument values. If a program applies min to two integers, the result will be an integer. This information can be

used to optimize the code that calls min. In general, this *type matching* can combine with a language's type generators, so that the type of one argument must match the type of a field of a record argument, and so on.

(As has been mentioned, the meaning of "the same type" depends upon the programming language. In a language with an assignment operation that copies values into variables, "the same type" may mean "the same representation". In another language, two values a and b might have the same type if a single "$<$" routine exists such that $a < b \Rightarrow \neg(b < a)$.)

These points illustrate two, more general forms of expressiveness: in the first case, the ability to describe properties of a type; in the second, the ability to describe relationships between types.

Within a program, interfaces of implementations of notions must express these properties. Perfectly *precise interfaces* state all of the notion's properties, and no others. A language that lets programmers define more precise interfaces is more expressive. An interface may fall short of perfect precision in two ways. It may state extra properties: in these cases, the language lacks *generality* (the ability to define interfaces and implementations with the largest possible domain of applicability possible), and some semantically well-defined computations are inexpressible. Alternately, interfaces may be *permissive*: they may omit relevant properties, in which case uses of the implementation can not be checked as carefully; in the best cases, potentially static checks must be replaced by explicitly programmed dynamic checks. (In the worst cases, the checks never occur. A "convention" comment that states what must or must not be done with some implementation is evidence of such a lack of precision; such unenforced conventions are dangerous, because subtle errors often result when they are accidentally or intentionally violated.)

Separate Interfaces The definition of an implementation always defines an interface of that implementation. Some programming languages provide no other way to define interfaces; in others, the implementation and its interface must reside in one textual unit. The designers of Ada [57] took a different approach. Ada source code is gathered into units called *packages*, which are divided into two parts: the *package specification* defines the interface of a package, and the *package body* provides the implementation. Ichbiah [29] argued that this design has the following benefits:

- Client programmers can be prevented from reading the implementation. This lets implementors keep the implementation confidential, for instance by distributing only a compiled form of the implementation along with the textual form of the interface.

- Even if confidentiality is not required, preventing client programmers from reading the textual form of the implementation keeps them from inferring, and making use of, incidental properties of the implementation that might be changed in the future. In other words, the interface provides a contract that clients and implementations can be measured against.
- Implementations and interfaces can be separately compiled safely.

I would like to add three points to this list:

- Translators only need access to interfaces, instead of the entire implementation, when checking client programs.
- It becomes easier to have several implementations corresponding to an interface, each suited to different clients.
- It becomes possible to have several interfaces to one implementation. One reason would be to provide interfaces corresponding to different levels of encapsulation, as in C++'s `public` and `protected` class interfaces [55]. Another is to provide interfaces for different sorts of clients; for instance an implementation of a buffer might provide "reader" and "writer" interfaces.

Code Reuse As the programming community develops experience in a problem area, it discovers new useful notions, and borrows old ones from other areas. These notions will be used in many programs. For instance, the proportion of commercial data processing code that is made up of such reusable notions is estimated to be 60% [32], or even 90% [4]. Hence, a programming language should encourage production of reusable code, since reuse can greatly reduce the cost of programming; Lanergan and Grasso believe that their approach to code and design reuse increase productivity by 50% when new programs are written and greatly eases the task of program comprehension during maintenance, once programmers have become familiar with the reusable code [32].

It is also important to provide support for different forms of reuse. The implementation of a notion can be used in many programs that need that notion, an implementation can be reused in the construction of similar implementations, and an interface can be reused in the description of similar interfaces.

The simplest way to reuse code is to copy and modify it. This is not a satisfactory form of reuse, since maintaining many copies of similar code increases wasted effort and raises the probability of errors and inconsistencies.

Flexibility obviously encourages reuse. So does the separation of interfaces from implementations. A client that can not make use of hidden details is protected from changes in the implementation of those details; this weakens the urge to use copy-and-modify as a defense against such changes. The stronger the separation is, the more it will encourage reuse.

The ability to replace an interface or implementation with a more general version, without affecting clients of the original, is called *generalizability*. Generalizability lets a programmer replace a "minimum" routine defined for integer parameters with one defined for parameters with any ordered type without requiring changes to any routine calls. This promotes reuse by reducing the fear of "ripple effects" that can result from changes to reused code.

The ability to augment the domain of an interface or implementation without modifying it is called *incrementality* [16]. Incrementality lets a programmer define a "less than" routine for a type that does not have one, perhaps so that instances can be passed to a "minimum" routine. This promotes reuse by allowing programmers to adapt existing implementations to new domains.

Simple Base Language "The language should get as much mileage as possible out of its definitional mechanism, never introducing something as a distinct language construct which can better be explained in terms of the definitional mechanism" [27, p. 13]. Replacing programming-language primitives with programmer-definable facilities leads to a smaller, simpler language, and simplicity is generally held to be a virtue in programming languages (provided that simplicity is not achieved at the expense of other desirable properties) [28, 65]. Furthermore, the existence of a standard library of useful facilities written in a programming language provides evidence of the flexibility and expressive power of the language.

Shaw and Wulf [49] argue that programming languages pre-empt many decisions that could be left open for programmers, and that this can result in contorted programs, can prevent optimization, and (because of lost efficiency) can discourage the use of high-level languages. They describe ways for programmers to retain control over concurrency, iteration, storage layout, and even procedure invocation. For instance, programming languages often provide some way to dynamically manage objects in memory. In the C language, this capability is provided by standard library routines such as `malloc` and `free` [2]. Programmers can replace these routines if they need tracing or debugging features [7, 67], or garbage-collecting allocators [5], or allocators that are tuned to their programs. In Pascal [30], storage management is provided by primitive `new` and `dispose` routines. Since they are tied in to the compiler's run-time environment, replacing them is usually difficult or impossible. In this respect, C is more flexible than Pascal. (Unfortunately,

C's type system is such that the flexibility is bought at the cost of a loss of type safety.) As another example, Hilfinger shows that, with some extensions to Ada's abstraction mechanisms, its exception handling and tasking facilities could have been provided by library packages [27].

1.3 Polymorphism

The type of a notion is often quite vague. Consider the notion of a "minimum" operation, which returns the smallest of its arguments. This includes notions of operations that take numeric arguments, or arguments from any ordered type. Similarly, the operation might take two arguments, or an unbounded number of arguments, or a single argument that is a set of such values. Hence, neither the types of the arguments nor the type of the notion "minimum" are rigidly defined.

Polymorphism allows languages to reflect this vagueness. Strachey informally defined polymorphic routines and operations to be those that "have several forms depending on their arguments" [54], and gave as an example the "+" operation, which in most programming languages works with many combinations of numeric types. He also coined terms for two main sorts of polymorphism: *ad-hoc polymorphism* and *parametric polymorphism.*

> In *ad-hoc* polymorphism there is no single systematic way of determining the type of the result from the type of the arguments. There may be several rules of limited extent which reduce the number of cases, but these are themselves *ad-hoc* both in scope and content. All the ordinary arithmetic operators and functions come into this category. It seems, moreover, that the automatic insertion of transfer functions by the compiling system is limited to this class.
>
> Parametric polymorphism is more regular and may be illustrated by an example. Suppose `f` is a function whose argument is of type α and whose result is of type β (so that the type of `f` might be written $\alpha \Rightarrow \beta$), and that `L` is a list whose elements are all of type α (so that the type of `L` is α `list`). We can imagine a function, say `Map`, which applies `f` in turn to each member of `L` and makes a list of the results. Thus `Map[f,L]` will produce a β `list`. We would like `Map` to work on all types of list provided `f` was a suitable function, so that `Map` would have to be polymorphic. However its polymorphism is of a particularly simple parametric type which could be written
>
> $$(\alpha \texttt{ list}, \alpha \Rightarrow \beta) \Rightarrow \beta \texttt{ list}.$$
>
> where α and β stand for any types.

The polymorphism exhibited by "the ordinary arithmetic operators" might be what is now called *overloading*, which allows several routines with distinct types but with the same name to exist within a scope. For instance, Algol 68 [58] defines two `not` operations, one for boolean values and one for bit strings. It also could cover what Strachey called *dynamic type determination*: arguments contain a type tag, and polymorphic routines dynamically test the tag to decide what to do.

Cardelli and Wegner introduced the term *inclusion polymorphism* for the sort of polymorphism found in "object oriented" languages, where "an object can be viewed as belonging to many different classes that need not be disjoint; that is, there may be inclusion of classes" [11]. They do not define the term "class", which means different things in different object oriented languages[1]. They group parametric and inclusion polymorphism together as forms of *universal polymorphism*. Inclusion polymorphism is associated with polymorphic data structures, while parametric polymorphism is associated with polymorphic routines.

These definitions are not precise, and the examples of *ad-hoc* polymorphism show that problems arise when they are applied to concepts that developed later. Does not the type checking algorithm of any compiler or run-time system constitute a "single systematic way" of determining the types of expressions in its language? How "regular" must a rule be to exclude a polymorphism mechanism from the *ad-hoc* category? Clearly it would be nice to have sharper definitions. One interpretation emphasizes the *behaviour* of polymorphic routines:

> *ad-hoc* polymorphism occurs when a function is defined over several different types, acting in a different way for each type ...parametric polymorphism occurs when a function is defined over a range of types, acting in the same way for each type. [60]
>
> Parametric polymorphism is obtained when a function works uniformly on a range of types; these types normally exhibit some common structure. *Ad-hoc* polymorphism is obtained when a function works, or appears to work, on several different types (which may not exhibit a common structure) and may behave in unrelated ways for each type ...In terms of implementation, a universally polymorphic function will execute the *same* code for arguments of any admissible type, whereas an *ad-hoc* polymorphic function may execute *different* code for each type of argument. [11]

This view has two (related) difficulties. First, Strachey seems to be concerned only with *types*, not with behaviour. Second, when the implementation of a polymorphic routine is hidden from

[1] "Class" is not the same as "type", which they do define.

clients, "the way it acts" (and especially the code it executes!) is unobservable. It seems better to define polymorphism only in terms of types and interfaces; that is, the "common structure" of arguments that is referred to above.

Note also that Strachey's terms apply to language mechanisms, not to the notions that are implemented with them. A language can provide more than one polymorphism mechanism; a notion might be implemented by an overloading of parametrically polymorphic and monomorphic routines, and in that case it would not make sense to say that the *notion* (or its implementation) exhibits parametric or *ad-hoc* polymorphism. The relevant property is the range of use of the implementation: is it applicable to some finite set of currently-defined types, or can it be used with an infinite set of types, including some that will be defined in the future? In the latter case, the types can not be specified directly; instead, the implementation of the notion must state some common structure or interface that the types must possess.

The following definitions are reasonably precise, emphasize the importance of common structure and infinite applicability, and refer to programming-language phenomena such as implementations and identifiers, not to notions or to unobservable phenomena such as the behaviour of implementations.

- *polymorphism* is the ability to implement a notion so that it applies to more than one type.
- *ad-hoc polymorphism* is present when an implementation has one or more interfaces that are defined for a set of types which need not have any common structure.

  This definition does not require any similarity in the types (or even the number of arguments) of the interfaces associated with an identifier, because it must cover the case of a "minimum" notion that has interfaces defined for a single (set-valued) argument and for a pair of arguments. Unfortunately, this also covers cases where the interfaces do not represent the same notion. Many programming languages have a binary "-" operator, that represents the notion of subtraction, and a unary "-" operator that represents the notion of arithmetic negation.
- *universal polymorphism* is present when an implementation has a single interface that is applicable to a potentially infinite set of types defined by a common structure.
- A routine exhibits *parametric polymorphism* when the value of a parameter defines other parts of the routine's interface. The parameterized interface defines a structure, and the routine accepts any argument list (and only those lists) that possess that structure, but

the lists can have a potentially infinite number of different types, corresponding to different parameter values.

- *inclusion polymorphism* is present when an identifier can be bound to values from a potentially infinite set of types with a common structure.

  The boundary between inclusion polymorphism and parametric polymorphism is thin. Consider the α `list` parameter of Strachey's `Map` example. It can be bound to values of different types during different calls, but it does not demonstrate inclusion polymorphism because its value can only have one type: α `list`, for the current value of α. If inclusion polymorphism was present, α `list` would only define some, but not all, of the attributes of the value's actual type, and the parameter could be bound to a list of any type that possessed α's attributes.

It is interesting to consider some boundary cases of these definitions. The language EL1 [63] uses the type generator `ONEOF(t1,t2)` to define the union of the types `t1` and `t2`. A routine that doubles an integer *or* a real number could be defined as follows.

```
DOUBLE1 ← EXPR( V:ONEOF(INT, REAL); ONEOF(INT, REAL))
    BEGIN
        V + V
    END
```

Then `DOUBLE1(3)` would return a `ONEOF(INT, REAL)` value containing the integer 6, and `DOUBLE1(3.5)` would return one containing the real number 7.0. By the "behavioural" definition, `DOUBLE1` is an example of parametric polymorphism because the routine body does not contain special cases for the different argument types. By the definitions used here, `DOUBLE1` is an example of *ad-hoc* polymorphism, because it does not use a required structure to define the types of arguments it accepts; it just lists a set of types. Both argument types have a "+" operation, but that does not define the set of legal argument types: `DOUBLE1` will not accept a `COMPLEX` argument, even though `COMPLEX` has a "+" operation.

EL1 also contains a `GENERIC` construct that allows routines to perform calculations based on the dynamic types of arguments.

```
DOUBLE2 ← EXPR( V:ONEOF(INT, REAL); ONEOF(INT, REAL))
    GENERIC(V)
        [INT] ⇒ FIXADD(V, V)
```

```
        [REAL] ⇒ FLOATADD(V, V)
    END
```

`DOUBLE2` is an example of *ad-hoc* polymorphism by any definition, including Strachey's. However, `DOUBLE1` and `DOUBLE2` have the same interface!

Chapter 2

Polymorphism Mechanisms

Chapter 1 discussed a number of desirable properties of programming languages, and defined several broad classes of polymorphism. This chapter discusses specific mechanisms that have been used to introduce polymorphism into languages, and evaluates the extent to which they support for the desirable properties. It does not deal in great detail with deep semantic issues; instead it concentrates on the pragmatic effects that these mechanisms have on programs that use them.

2.1 *Ad-Hoc* Mechanisms

Recall the definition given in chapter 1: *ad-hoc* polymorphism is present when an implementation has one or more interfaces that are defined for a set of types which need not have any common structure. *Ad-hoc* polymorphism comes in three varieties: overloading, transfer functions, and set-theoretic unions.

2.1.1 Overloading

"A construct in a typed programming language is said to be *overloaded* when there are several different implementations of the construct and the version appropriate to a particular case is chosen using type information reflecting its context within the program"[61]. For example, Algol 68 overloads the `not` operator: one version takes a boolean argument and returns a boolean value, and the other takes and returns bit strings. Overloading is an *ad-hoc* mechanism because each

implementation has a different interface, and the types mentioned in different interfaces need not have anything in common.

When an overloaded identifier is used, one of its implementations must be selected. Statically checked programming languages usually use *static overload resolution*: the language translator attempts to choose one implementation of each overloaded identifier in an expression, based on the type rules of the language and the declarations of identifiers in the context of the expression. If the translator can not choose between two implementations, it judges the expression to be ambiguous and rejects it. Consequently, a valid expression using overloading is equivalent to an expression that uses unique identifiers, so static resolution has no effect on generalizability, static checking, strong checking, or separation of interfaces from implementations. It is also efficient, since there is no added run-time cost, and since the increased compilation cost should also be acceptable; arguably, if a language definition makes overload resolution expensive, then overloading in that language is too complex to be used effectively by programmers.

Overloading provides incrementality, since more overloadings can be added if they are needed. It simplifies the language, since notions like `not` can be presented as a library instead of a special case of the language. It also increases expressiveness by reducing the severity of name clashes: for instance, two libraries can declare the same identifier, as long as the overloading rules allow them to be distinguished. It provides type matching: consider two implementations of `min`, one accepting a pair of integers and returning an integer, and the other accepting and returning real numbers. The arguments and result must have the same type. Overloading also lets interfaces define their applicable notions, to a limited extent. However, it does little to increase generality; a full description of the notion of "minimum" would require an infinite number of overloadings.

Other drawbacks of overloading are due to its unconstrained nature. First, there is no guarantee that all overloadings of an identifier implement the same notion, and hence a small programmer error can result in a call to a routine that does entirely the wrong thing. Second, many authors feel that overloading can reduce program readability [23, 59]. This is partly because, in a case where an identifier represents more than one notion, it can be difficult to determine which notion the author of some code had in mind. But it is also due to a paradoxical effect: overloading makes it easier to get a general understanding of code ("Aha! This code sums the elements of a vector!"), but makes it harder to get a detailed understanding ("Aha! This code calls the addition routine for quaternions!") These problems can be reduced if programmers have access to program analysis tools, and if programmers use taste and discretion when overloading.

2.1.2 Transfer Functions

One of Strachey's examples of *ad-hoc* polymorphism was the automatic insertion of a *transfer function*. This is also known as *implicit conversion* and as *coercion*. Cardelli and Wegner define it as "a semantic operation that is needed to convert an argument to the type expected by a function, in a situation that would otherwise result in a type error [11]". For instance, the `sin` routine in Algol 68 takes a `real` argument. If it is called with an integer argument, the language translator inserts a transfer function to convert the argument to a `real` value, and passes the result to `sin`. In languages such as C++, programmers can define new transfer functions to complement programmer-defined types.

Note that, by this definition, a transfer function creates a new value of the desired type. In certain cases, the bit patterns of the old and new values may be the same and no computation is needed, but that is an optimization. (By some definitions, coercions merely change the type of an existing value, without creating a new value or altering it in any way.) A call to a `sin` routine that takes a `real` parameter will always return a `real` result, even if the argument is an integer. If an integer result is needed, then another transfer function must be applied to the `real` result to create it. In general, such inverse transfer functions may not exist, or successively applying the transfer function and its inverse may result in the loss of information present in the original value.

Transfer functions introduce polymorphism because routines like `sin` can be applied to many different types of arguments. The polymorphism is *ad-hoc* because the set of types is defined by the set of available transfer functions, not by any properties of the types. It is different from other polymorphism mechanisms in that it adapts the arguments to suit the routines; other polymorphism mechanisms produce routines that adapt to their arguments. Hence it has no effect on interface precision or separation of interfaces from implementations, and does not provide a way to simplify a language by replacing language constructs with programmer-definable abstractions.

An expression that uses transfer functions is equivalent to an expression where the transfer functions are called explicitly, so implicit calls do not affect strong static checking and are no less efficient than explicit conversions. Programmer-defined transfer functions provide a weak form of incrementality, since new transfer functions let old routines be applied to arguments of new types, and may occasionally provide some generalizability, if a new, more general routine is accompanied by a transfer function that converts old arguments to the new form. However, the potential loss of type precision and of information makes transfer function polymorphism unsatisfactory in most cases.

2.1.3 Set-Theoretic Union Types

The EL1 `ONEOF` type used by the routines `DOUBLE1` and `DOUBLE2` on page 12 is an example of a *set-theoretic union* type. More advanced versions of the idea have been proposed by Reynolds [45] and Pierce [43]. These types are defined in terms of other *member* types; if a type is considered to be a set of values, then the union type is the set of all values contained in those member types. Consequently a value may belong to more than one type. Set-theoretic unions provide polymorphism, because they allow identifiers to be bound to values from the different member types. The polymorphism is *ad-hoc* because no common structure is needed among argument types.

The `GENERIC` construct used by `DOUBLE2` is an example of a *conformity clause*. A conformity clause is similar to a `case` statement, but it examines the type of a data item instead of its value. Each branch of the clause describes types that the data item might have, and statements to execute if it has one of those types. If the branch specifies a type constant, then strong static checking is possible in the branch.

A language might provide *dynamic overload resolution* instead of conformity clauses. In such a language, if an expression uses an overloaded routine identifier, and an argument to the routine is a union, then the member type that the value belongs to would be used to resolve the overloading. In effect, an implicit conformity clause is used to call the appropriate implementation of the overloaded routine. The overloaded routines in such languages are sometimes called *multimethods*. Ghelli has described a strongly and statically checked type system for such a language [22].

Many languages provide unions that are not set-theoretic unions. In these languages a union type is distinct from its member types, and special actions are needed to convert between values of the union type and values of the member types. Consider the Algol 68 version of `DOUBLE2`.

```
mode ir = union(int, real);
proc double68 = (ir v) ir:
    case v
    in (int i): i + i,
       (real r): r + r
    esac;
double68(3);
double68(3.5)
```

Note that the `case` statement (the Algol 68 conformity clause) provides a name (i or r) to refer

to the member-type value contained within the union parameter *v*. *v* can not be assigned to a `real` variable or passed to a `real` parameter, even if it happens to contain a `real` value. This shows that there is a distinction between a union value and the member value that it contains. When a member value such as `3` is passed to *double68*, a *uniting coercion* implicitly converts it to a union value. Hence, the polymorphism of `double68` is actually transfer function polymorphism, not union polymorphism.

Unions increase expressiveness in an interesting way: it becomes possible to handle data whose exact type is not statically known. For instance, it may be impossible to statically decide whether the argument of a call to `DOUBLE1` is an `INT` or a `REAL`: the argument might be a `ONEOF` parameter of a routine that calls `DOUBLE1`, or its value might depend on input to the program.

Unions provide a small amount of generalizability, since a union type can be expanded to include more members: `DOUBLE1` could be changed to have a parameter of type `ONEOF(INT, REAL, COMPLEX)` without affecting its clients. They allow language simplification; for instance, polymorphic operators can be presented as library routines. They have no effect on incrementality or on separation of interfaces and implementations. As an example of interface precision, consider an EL1 "minimum" routine:

```
MIN ← EXPR( A: ONEOF(INT, REAL); B: ONEOF(INT, REAL);
    ONEOF(INT, REAL))
    BEGIN ... END
```

This example shows that unions prevent type matching: one argument of `MIN` could be an `INT` while the other is `REAL`. Furthermore, if the arguments have the same type, the value returned by `MIN` has that same type, but `MIN`'s type does not express that fact. The example also shows that unions do not provide full generality, since the arguments to a fully general `MIN` routine would have to be unions of an infinite number of ordered types.

Set-theoretic unions can have reasonable efficiency. Typically, types are encoded as type tags. Distinct types must have distinct tags. (This may be difficult to arrange for some combinations of computing environments and type systems.) Type tags have a space cost; in the worst case, every value must contain a type tag, but a translator can often optimize away the tags of values whose types are never in doubt. Since the exact size of a union value is unknown, enough space must be allocated to hold the largest possible value, or values must be manipulated via pointers.

The execution time cost of type tags is moderate; Steenkiste [53] studied an implementation for the MIPS-X processor of Portable Standard Lisp, a dynamically-typed language. He concluded that a collection of benchmark programs would spend 22% to 32% of their time processing tags.

2.2 Universal Mechanisms

The common feature of all *ad-hoc* polymorphism mechanisms is that the domains of implementations that use them are not defined by some common structure. In contrast, universal polymorphism uses a common structure to define a single interface that is applicable to a potentially infinite set of types. It has two varieties: inclusion polymorphism lets programs bind identifiers to values from a set of types defined by a common structure, and parametric polymorphism lets the interface of a routine depend on the value of one of its parameters, so that valid argument lists must match a structure defined by the interface. Several polymorphism mechanisms exist for each of these varieties of polymorphism.

2.2.1 Inclusion Polymorphism

Some programming languages define a *subtype* relationship between types. If a type τ_1 is a subtype of another type τ_2 (written $\tau_1 \subseteq \tau_2$) then values of type τ_1 can be used in certain situations that require a value of type τ_2. The situations vary among languages, but typically they use the following type-checking rule:

$$\frac{e : \tau_1 \qquad \tau_1 \subseteq \tau_2}{e : \tau_2}$$

Combining this with the function-checking rule on page 3 gives

$$\frac{e_1 : \tau_2 \rightarrow \tau_3 \qquad e_2 : \tau_1 \qquad \tau_1 \subseteq \tau_2}{e_1(e_2) : \tau_3}$$

This is a universal polymorphism mechanism, because routines accept arguments with any of a set of types that share a common structure defined by the language's subtype rules. It is inclusion polymorphism because the parameters of the routine can be bound to arguments with different subtypes of the parameter type.

Subtyping can allow simplification of the language kernel. Recall that the Algol 68 `not` operator performs logical negation on boolean and bit string values. In a language with subtypes, this could be expressed as a single routine that takes an argument of type "logical", which would be a supertype of the boolean and bit string types.

If every operation that is valid for values of a type is also valid for all subtypes of the type, then subtyping preserves strong static checking. Subtyping has no effect on separation of implementations and interfaces. However, it does reduce the precision of interfaces, since it leaves no static way to refer to the actual type of an argument to a polymorphic routine; only the name of the supertype is known. Hence type matching vanishes: there is no way to declare statically that two values have *exactly* the same type. Such information, if present at all, is given in comments, and enforced at run-time, when it is too late to correct errors.

Subtypes can also provide polymorphic data structures, for instance if the language allows record fields declared to have type τ to be bound to values of subtypes of τ.

Infinite Unions

The simplest form of subtyping defines a type that is a supertype of every other data type, or in other words, an infinite set-theoretic union of all other types. (It is possible to have infinite versions of Algol 68's style of union, but they are not discussed in this section.) For instance, EL1 has an `ANY` type, which can be combined with the type generators of the language to produce records with fields that can hold any value, parameters that accept any argument, and so on. Routines that have parameters whose types involve infinite unions exhibit inclusion polymorphism; the "required structure" of the argument types is the set of properties common to all data types, such as the ability to be assigned or instantiated. If an infinite union is used in a type generator, then the generator defines a more precise structure: the EL1 routine

```
NAME_OF ← EXPR( S:STRUCT( NAME: STRING, VAL: ANY );
                STRING)
    BEGIN
        S.NAME
    END
```

will accept as an argument any record that has two fields, the first of which is a string called `NAME`, and the second of which is anything called `VAL`.

The implementation of infinite unions can be much like the implementation of finite unions, except that when allocating space for an infinite union variable, a compiler can not simply allocate enough space to hold the largest member type. Consequently, indirection must be used in more cases.

Strong interface checking and static checking can be applied in the cases of operations that can be performed on values of type `ANY`, and of operations that can be performed on every data type: a typical example is computing a reference to a value. These operations are surprisingly useful. For example, routines that implement operations on "collection" types such as lists often use only those operations, since the routines reorganize values but do not operate on them directly. Hence these languages allow safe, statically checkable implementations of collection types that are independent of the collection's element types. This is an increase in both expressiveness and flexibility, and therefore in reusability.

```
ADD ← EXPR( C:COLLECTION, V: ANY) ...
```

Unfortunately, due to the absence of type matching, static type information about data that passes through polymorphic routines or data structures vanishes. (A program can add an `INT` to a collection, but when some operation retrieves the integer from the collection, its static type will be `ANY`, not `INT`.)

Infinite unions allow smaller language kernels, since operations like the traditional `cons`, `car`, and `cdr` operations on lists can be treated as library routines. However, routines can only replace language primitives that take any value as an argument: a `not` operation should not be defined this way, because "logical negation" is not a sensible operation for most types.

When a data item has static type `ANY`, its dynamic type of the data must be recovered before any type-specific operation can be performed on it. This could be done with a conformity clause, but the usefulness of this has limits because only a finite number of tests can be performed. Checking could also involve dynamic overload resolution, with its run-time costs and dynamic checking. Another alternative is to pass the operations along with the data, as routine parameters or as routine fields of the value. This requires some care on the programmer's part, and still requires dynamic type checking in general.

Infinite unions provide a form of generality, but not a useful form, because they do not allow programmers to specify many sorts of restrictions on the values that a union may take on. For instance, a truly polymorphic version of `DOUBLE1` should accept any type that has a "+" operation, but the `ANY` type does not provide a way to specify that *only* such values are acceptable. Similarly, it provides only weak generalizability (a routine with an `INT` parameter can be replaced by one with an `ANY` parameter) and incrementality (any new type is a subtype of `ANY`, and hence extends the domain of routines with `ANY` parameters).

Record Subtyping

Cardelli [1, 9] described subtype rules for routine and record types. A subtype of a record type can add new fields, and can change the type of an existing field to a subtype of the original field type:

$$\frac{\tau_1 \subseteq \tau_1' \qquad \tau_2 \subseteq \tau_1' \qquad \cdots \qquad \tau_m \subseteq \tau_m'}{\langle l_1 : \tau_1 \times \cdots l_m : \tau_m \rangle \subseteq \langle l_1 : \tau_1' \times \cdots l_m : \tau_m' \times \cdots l_n : \tau_n' \rangle}$$

In effect, this rule provides a looser version of structural equivalence for record types.

The rule for routines also allows changes in types: a subtype of a routine type can replace the result type by a subtype, and can replace the argument type by a supertype.

$$\frac{\sigma_1 \subseteq \sigma_2 \qquad \tau_1 \subseteq \tau_2}{\sigma_2 \rightarrow \tau_1 \subseteq \sigma_1 \rightarrow \tau_2}$$

For instance, if a program uses two types `Integer` and `Fraction`, with `Integer` $\subseteq$ `Fraction`, then

`Fraction` $\rightarrow$ `Integer` $\subseteq$ `Integer` $\rightarrow$ `Fraction`

and a routine with the first type can be passed as an argument to a parameter with the second type. This pattern of replacement is known as *contravariance*, since the result and argument types change in opposite ways. It follows naturally from the flow of data through routines: in through the parameters and out through the result, with subtype-to-supertype transformations at each point.

Unfortunately, many operations are naturally *covariant*. Consider the natural types of the addition operations associated with `Fraction` and `Integer`:

`+: Integer` $\rightarrow$ `Integer` $\rightarrow$ `Integer`

`+: Fraction` $\rightarrow$ `Fraction` $\rightarrow$ `Fraction`

Since these operation's types exhibit covariance, neither's type is a supertype of the other's, whether `Fraction` and `Integer` have some subtype relationship or not. Furthermore, there can be no type `T` such `T`$\rightarrow$`T`$\rightarrow$`T` is a supertype of both. This makes it impossible to write a "double" routine that would accept either "+" operation as an argument. This problem can not be solved by using a covariant subtype rule, because covariant rules require dynamic type checking [13]. The best that can be done would give the two routines the types

`+: Addable` $\rightarrow$ `Addable` $\rightarrow$ `Integer`

`+: Addable` $\rightarrow$ `Addable` $\rightarrow$ `Fraction`

where `Addable` is some supertype of `Integer` and `Fraction`; then both types would be subtypes of `Addable`→`Addable`→`T2`, if `T2` is a subtype of both `Integer` and `Fraction`. Clearly this approach would lose type precision. It also would require two problematic types, `Addable` and `T2`, which must be a supertype and a subtype, respectively, of all types with a "+" operation.

Record subtype languages have a second distinguishing feature: routines stored in fields in records can refer directly to the record itself, typically through the special identifier `self`. In effect, the record becomes an extra parameter of the routines that it contains. Programs that use record subtyping would not implement "+" as an ordinary routine. Instead, programmers implement types such as `Fraction` as records that bundle together a value and the routines that operate on it. Built-in types such as `Integer` are treated as if they were records containing routines. Polymorphic routines use record types as parameter types, and accept any argument whose type is a subtype of the parameter type.

```
type Addable = ⟨plus: Addable → Addable⟩
-- assume Integer ⊆ Addable
type Fraction = ⟨plus: Addable → Fraction ×
                 numerator: Integer × denominator: Integer⟩
-- Fraction ⊆ Addable.

double: Addable → Addable = ...
-- Double the fraction "one half"
double ⟨numerator := 1, denominator := 2,
        plus:=λp:Addable·(... self.i ...) ⟩
```

This does not prevent the loss of static type information mentioned above: the `plus` routine for `Fractions` still takes an `Addable` argument, not a `Fraction`, and the result of the call to `Double` has static type `Addable`, not `Fraction`. However, this is an improvement over the situation presented by infinite union polymorphism, since routine and record types refer to `Addable`, not to an overly-general type like `ANY`. Furthermore, the subtype rules let subtypes change the types of routine fields, as long as covariance is observed: the result type of the `plus` field changes from `Addable` in the declaration of `Addable` to the more precise `Fraction` in the declaration of `Fraction`. Consequently, programs have less need for conformity clauses or dynamic overload resolution.

Subtyping provides generality: a routine will accept any argument that supports the operations defined by the routine's parameter's fields. However, the contravariant subtyping rule for

routines has the effect of limiting generality, as shown in [8] (where the problem arises in "negative recursion" in routine fields of recursive record types). Consider this attempt to define a polymorphic `min` routine, which uses the notion of "ordered" values: values that provide a `lt` operation that returns `true` if the value is less than or equal to `lt`'s argument.

```
type Ordered = ⟨lt: Ordered → Boolean⟩
min: Ordered → Ordered → Ordered
   = λp:Ordered·λq:Ordered·if p.lt(q) then p else q
```

(`Ordered` serves a useful purpose, even though instances of it contain no data and its `lt` routine field has not been implemented; it defines the interface of ordered values.) A programmer might define a `String` type, containing character strings, with a `lt` operation that compares strings in lexicographical order:

```
type String = ⟨... lt: String → Boolean ...⟩
```

However, `min` does not accept `String` arguments, even though `String` defines `lt`, because `String` is not a subtype of `Ordered`. This can be demonstrated by contradiction: assume

$$\texttt{String} \subseteq \texttt{Ordered}.$$

Then the record subtyping rule and the types of their `lt` fields gives

$$\texttt{String}\rightarrow\texttt{Boolean} \subseteq \texttt{Ordered}\rightarrow\texttt{Boolean}$$

and applying the routine subtype rule to the parameter types gives

$$\texttt{Ordered} \subseteq \texttt{String}$$

Combining this with the original assumption gives

$$\texttt{Ordered} = \texttt{String}$$

Which is clearly false. `String` would be a subtype of `Ordered` if it were defined as

```
type String = ⟨... lt: Ordered → Boolean ...⟩.
```

In this case, `String` $\subseteq$ `Ordered`, but `lt`'s parameter type is too general, and gives `lt` no fields with which to do the comparison! In cases like these, subtyping languages can not define their parameter types with full generality.

Note that routine types and record types can not have a subtype relationship; hence a polymorphic routine can not have a parameter that can accept both routine arguments and data arguments. There are some notions for which this makes sense (see [16] for an example), but these notions can not be implemented in their full generality.

The loss of type matching has one potentially beneficial consequence: it allows "mixed mode" operations. Since `Integer` and `Fraction` are both subtypes of `Addable`, the type-checking rules let programmers add integers to fractions. This is not an unmixed blessing.

- The implementation of `plus` for fractions must do something sensible for any argument type that is a subtype of `Addable`, even for types created after `Fraction`.
- Assume that the programming language provides a type that implements the real numbers and is a subtype of `Addable`. The result of adding a fraction to a real number is a real number; in general, for any `T`$\subseteq$`Addable`, the result of adding a `T` to a fraction might be a `T`, so `Fraction`'s `plus` must have type `Addable`$\rightarrow$`Addable`, not the more precise `Addable`$\rightarrow$`Fraction`
- Two types can have a common supertype, and yet mixed mode computations between them may not make sense.

The last point requires an extended example. As an example, consider a program that also uses a `Name` type that is a subtype of `Ordered`, but that has a `lt` routine that compares names in "phone book" order instead of lexicographical order.

```
type Name = ⟨... lt: Ordered → Boolean ...⟩
s: String = ...
n: Name = ...
min s n
min n s
```

Both calls are legal: `Name` and `String` are both subtypes of `Ordered`, so `s` and `n` are legal arguments given the subtype-based type-checking rule. However, the calls could have different results: "St. John" comes before "Smith" in phone books. Since subtyping weakens the property of type

matching, the programmer has no way to specify that the two types are actually incomparable. This disagreement is arguably minor where `min` is concerned, but a sorting routine given a mixture of `String`s and `Name`s might fail to terminate if this sort of ambiguity caused it to repeatedly swap a `Name` value and a `String` value with each other.

A type defines operations that every subtype must provide, and hence represents a notion that contains all of those subtypes. The name of the field provides a statically checkable (but weak) indication of the semantics of the operation: a routine field named "plus" probably implements addition. Record values implement the notion, and the record's routine fields implement implement operations associated with that notion. For example, a program that performs geometric calculations might use several types representing regions of the Cartesian plane, each of which provides a `contains` operation that decides if the region contains a given point. The notion of "region" can be represented as a type.

```
type Point = ⟨x:Real × y:Real⟩
type Region = ⟨contains: Point → Boolean⟩
in_both: Point → Region → Region → Boolean
       = λp:Point·λr1:Region·λr2:Region·
           (r1.contains p) ∧ (r2.contains p)
```

Rectangles and circles are regions.

```
type Rect   = ⟨top_left:Point × bottom_right:Point
            × contains: Point→Boolean⟩
type Circle = ⟨origin:Point × radius: Real
            × contains: Point→Boolean⟩
a_rect: Rect = ...
a_circ: Circle = ...
a_pt: Point = ...
b: Boolean = in_both a_pt a_rect a_circ
```

Subtyping also provides polymorphic data, since identifiers can be bound to instances of subtypes of their declared types.

```
r: Region = if b then a_rect else a_circ
... r.contains a_pt ...
```

The call to `r.contains` is statically type-safe, because all regions have a `contains` field, but the value of `r` determines whether the circle's or rectangle's routine gets called.

Subtyping provides generalizability: a routine can be replaced by a routine whose argument types are supertypes of (and hence more general than) the original's types. It increases incrementality: any new type extends the domain of routines that accept its supertypes as arguments. Incremental extension does not work so well with existing types. Consider a record type `T` which needs to be extended with an operation `o` to become a subtype of some other type. Writing a routine called `o` with a `T` argument is not sufficient, because that does not change any subtype relationships. Adding a new `o` routine field requires control of the definition of `T`. Defining a new type `T2` containing a `T` instance and an `o` routine field requires `T`-to-`T2` conversions wherever the program uses an existing `T` value.

Statically Typed Object-Oriented Languages

Statically typed object-oriented languages make use of various forms of record subtyping. SIMULA [18, 52], C++ [19], Eiffel [34] and Modula-3 [10] are typical of these languages. Such languages generally refer to a record type as a *class*, a routine field as a *message*, and an implementation of a routine field as a *method*. Calling a routine field of a record is often called *message passing*. Typically, classes do not just define record types; they also define the implementations of the routine fields. (In the simplest forms of record subtyping, record types merely declare routine fields, to which routines are assigned when instances of the record are created.)

Programmers can create class definitions incrementally through *inheritance*, which creates a new *child class* from explicitly named *parent classes* by copying their definitions and possibly adding, hiding, deleting, or replacing field definitions. A class's parents, grandparents, etc., are collectively known as its *superclasses*, and its children, grandchildren, etc., are its *subclasses*. Allowing a class to have more than one parent introduces many complications into a language, and goes by the special name of *multiple inheritance*.

If a class has a routine field that does not have a defined implementation, it can not be instantiated, for fear that the instance's routine field might be called. Such a class is called an *abstract superclass*. An abstract superclass can act as an interface, for which non-abstract subclasses provide implementations. They can also provide guidance and assistance to the implementor of a new class, who can search for an abstract superclass with an appropriate interface, define the new class by inheritance from it, and implement the appropriate routines. An abstract `Collection` class might declare that it has `insert` and `remove` routines without implementing them; a `Stack`

subclass would define implementations that provide last-in, first-out behaviour. A *pure* abstract superclass provides nothing but unimplemented routine fields, and exists only to define an interface. Inheritance from a pure abstract superclass reuses *designs* instead of code. An *impure* abstract superclass implements some routines (in terms of other, implemented or unimplemented, routines), and provides both design and code reuse; the implemented routines assist programmers who must implement non-abstract subclasses.

Record subtyping rules were designed to provide clean semantics and a sound type checking system for "object-oriented" languages, so it is not surprising that these languages can be viewed as having a more or less restricted form of record subtyping. Many of these restrictions and complications result from attempts to combine three concepts: encapsulation, subtyping, and code reuse.

In C++ and Eiffel, classes provide encapsulation: clients of a class have access to some (but not all) of the fields of the instances of the class. The accessible fields make up an interface of the class. C++ classes have separate interfaces for clients that use class instances and for clients that are subclasses. Plain record subtyping provides encapsulation through static scope rules, not through record types: if a routine with return type `T` returns a value that is of a subtype of `T`, any extra fields defined by the subtype are inaccessible outside of the routine. This approach is somewhat more flexible because each such routine defines a different interface for different sorts of clients: the number of interfaces is not fixed at one or two.

Inheritance provides code reuse, since subclasses inherit the field definitions and routine field implementations of their superclasses unless they explicitly replace them. Multiple inheritance leads to the possibility of inheriting a declaration of a field or an implementation of a routine field from more than one parent. Different languages use different methods for resolving the resulting conflicts. Some of these methods expose the implementation of classes in subtle ways, so that changes in the implementation of a superclass force changes in subclasses or in clients [50]; this weakens encapsulation and makes reuse more difficult. A class may also inherit from a single superclass along more than one "line of descent", and this also introduces conflicting definitions. These problems do not arise with pure record subtyping, because record types are not constructed by inheritance.

Inheritance also affects subtyping, since it defines the child's *interface* in terms of the parent. In all four languages mentioned above, subclasses are automatically considered to be subtypes of their superclasses. As a result, redefinition of inherited fields must be restricted to ensure that subclasses truly are subtypes. (Eiffel uses a covariant rule for routine subtyping, and hence

is not statically type safe [13].) In SIMULA, Eiffel, and C++, subclasses are the *only* subtypes of superclasses. (If pure record subtyping provides a looser form of structural equivalence, then explicit superclass declarations provide a looser form of name equivalence.) This makes the subtype hierarchy explicit, and therefore rigid. If a program requires the definition of a new class that must be a superclass of existing classes, or if an existing class must be made a subclass of a class that is not one of its current superclasses, then existing inheritance declarations must be rewritten. Modula-3 is less restrictive: a class is a subtype of its parent or of any class that has the same fields as its parent (and, transitively, of their supertypes), so programmers can insert new classes into the class hierarchy without changing the definitions of old classes. Even this links inheritance to subtyping: the names of a class's supertypes are not fixed, but the fields that are inherited at each level are fixed.

Uniting inheritance with subtyping contorts programs. Consider an operating system where files are associated with significant data structures (perhaps containing data buffers, disk addresses, and such), and a program that implements notions of input files, output files, and input/output files with types `InFile`, `OutFile`, and `IOFile`. `IOFile` is a natural subtype of both `InFile` and `OutFile`, but the most convenient implementation would implement the file data structures completely in `IOFile`, and define the other two types by inheriting from `IOFile` and removing the extra operations. In this case, subtyping and inheritance relationships run in opposite directions.

2.2.2 Parametric Polymorphism

Universal Quantification

Some programming languages allow the type of a routine to depend on the *value* of some of its parameters. In the simplest case, an integer parameter might be used as an array bound in the declaration of another parameter. In the more interesting cases, routines can have *type parameters* that are used in the types of other parameters. Tennent [56] suggested adding them to Pascal (for reasons based on language design principles) and gave this example of a procedure that applies a routine to every element of an array, regardless of array bounds or element type.

```
procedure maparray(type ind, d;
                   var A: array[ind] of d;
                   procedure p(var :d)) =
     for i: ind do p(A[i]);
```

A typical use of maparray would be

```
procedure double(var i: integer) =
     i := i + i;
type ix: 1..10;
     ar: array[ix] of int;
var  a: ar;
begin ...
     maparray(ix, int, a, double);
     end
```

Various semantics have been proposed for languages with type parameters. The most popular is that of Girard [24] and Reynolds [44], who defined a typed lambda calculus where type parameterization is an abstraction mechanism distinct from, but similar to, functional abstraction. The identity routine on integers is written

$$\texttt{id\_int} = \lambda \texttt{p:Integer} \cdot \texttt{p}$$

so that "`id_int 3`" has the value 3. Similarly, the polymorphic identity routine is written

$$\texttt{id} = \Lambda \texttt{T} \cdot \lambda \texttt{p:T} \cdot \texttt{p}$$

and has the *universally quantified type* $\forall \texttt{T} \cdot \texttt{T} \rightarrow \texttt{T}$. The expression "`id Integer 3`" has the value 3 and type `Integer`.

Not all universally quantified types are sensible and useful. To see this, consider the declaration

$$\texttt{thing}: \forall \texttt{T} \cdot \tau(\texttt{T}) = \Lambda \texttt{T} \cdot e$$

where $\tau(\texttt{T})$ is some type expression possibly involving `T`, and e is some appropriately-typed expression. What values could e have? The key observation is that e must have type $\tau(\texttt{T})$ *for every conceivable type* `T`. This limits e to polymorphic identifiers applied to `T` and monomorphic values with no relevance to `T`. Hence

$$\texttt{thing1}: \forall \texttt{T} \cdot \langle \texttt{l:Integer} \times \texttt{r:Integer} \rangle = \langle \texttt{l := 3, r := 4} \rangle$$

is valid, since the `l` and `r` fields must contain integers, and 3 and 4 have type `Integer`.

```
id: ∀T·T → T = ΛT·λp:T·p
thing2: ∀T·⟨f: T → T⟩ = ⟨f := id T⟩
```

is also valid, since `f` must contain a routine of type `T→T` for every possible `T`, and "`id T`" has that type. However

```
pair: ∀T·⟨l:T × r:T⟩ = ΛT·⟨l := 3, r := 4⟩
```

is not legal, because `3` and `4` do not have type `T` for every possible `T`. Consequently, $\forall$`T·⟨l:T ×r:T⟩` can not be used as the type of a data structure containing a pair of values of unknown but identical type. In general, $\forall t \cdot \tau$ can not be used as the type of a polymorphic data structure containing a data item of type τ.

Instead, universal quantification provides polymorphic data in a more indirect form, by encoding it as a polymorphic routine [39, p. 39]. Recall the "region" example on page 26, with a `Region` type and a `Rect` subtype.

```
type Region = ⟨contains: Point → Boolean⟩
type Rect   = ⟨top_left:Point × bottom_right:Point
           × contains: Point→Boolean⟩
a_pt: Point = ...
a_rect: Rect = ...
r: Region = a_rect
... r.contains a_pt
```

In a language with universal quantification, `Rect` has a simple equivalent.

```
type Rect = ⟨top_left:Point × bottom_right:Point⟩
rect_contains: Rect → Point → Boolean = ...
```

In the subtyping example, binding `r` to `a_rect` had the effect of forgetting `a_rect`'s type. With universal quantification, the same effect can be created by hiding `a_rect` inside a routine. The routine's type is the equivalent of `Region`. Computations on the hidden data can be performed by packaging them as routines and passing them into the `Region` routine as arguments; the `Region` routine simply applies the computations to the data and returns the result. Since the type of the result is not known in advance, the `Region` routine must be polymorphic.

```
type Region = ∀res_t·(⟨contains:Point→Boolean⟩→res_t) → res_t
r: Region = Λres_t·λcomp: ⟨contains:Point→Boolean⟩ → res_t·
                comp ⟨contains := (rect_contains a_rect)⟩
```

To use `r`, a program must pass it the result type and a routine that performs the desired computation.

```
r Boolean (λregion:⟨contains:Point→Boolean⟩·
              region.contains a_pt)
```

Clearly, some "syntactic sugar" is needed to make this approach palatable.

Since a type parameter is a name for a specific type, it provides type matching: a two-argument `min` routine could be defined as

```
min: ∀T·(T→T→Boolean) → T → T → T
   = ΛT·λlt:T→T→Boolean·λa:T·λb:T·
      if lt a b then a else b
```

so that "`min Integer (<) 3 5`" would have the value `3`. Universal quantification also provides great generality: a routine's interface specifies exactly what types and operations it requires, and can be applied to any set of arguments that supplies those types and operations. Furthermore, these operation parameters do not have to follow the contravariant pattern required by subtyping languages: universal quantification has no difficulty specifying covariant patterns such as `T→T→T`.

Unfortunately, `min` must take the type `Integer` and the integer "less than" routine as an explicit parameter. Most routines having type parameters require several operation parameters that can be applied to instances of the type parameters, and perhaps some constants as well. Cormack and Wright point out that

> the existence of these extra parameters means that the greater the generality of an abstraction, the greater the burden of use on the programmer, and directly conflicts with our design criteria of generalizability and incrementality [16].

Consider a simple polymorphic "absolute value" routine would require a type argument, comparison and subtraction operations, and a "zero" value. Passing them as separate parameters leads to verbose routine calls and is a source of errors.

```
abs: ∀t·(T→T→Boolean) → (T→T→T) → T → T → T
   = Λt·λlt:T→T→Boolean·λsub:T→T→T·λzero:T·λx:T·
        if (lt x zero) then (sub zero x) else x
... abs Integer (<) (-) 0 3
```

Passing them as routine fields of record-type arguments is awkward in the absence of subtyping because the set of fields in the record argument type must be defined exactly; records must be disassembled and reassembled when programs pass data to different polymorphic routines that require different sets of operations.

```
min_rec: ∀T·⟨v:T × lt:T→T→Boolean⟩
            → ⟨v:T × lt:T→T→Boolean⟩
            → ⟨v:T × lt:T→T→Boolean⟩
       = ΛT·λa:⟨v:T × lt:T→T→Boolean⟩·
            λb:⟨v:T × lt:T→T→Boolean⟩·...
... min_rec Integer ⟨v:=3, lt:=(<)⟩ ⟨v:=5, lt:=(<)⟩
```

A polymorphic routine may also be too general for some situations; for instance, a programmer might want to pass it to a monomorphic parameter of another routine. In that case the programmer must create a *specialization* that binds some or all of the type and operation parameters:

```
fiddle: ∀T·(T→T→T) → T → T → T → T
      = ΛT·λf:T→T→T·λa:T·λb:T·λb:T·
           f a (f b c)
min_int: Integer → Integer → Integer
       = λx:Integer·λy:Integer·min Integer (<) x y
... fiddle Integer min_int 19 83 18 ...
```

Universal quantification does not affect separation of interfaces from implementations, since the name it introduces is in some sense "just another parameter". Strong checking is also unaffected. Incremental extension of data types is simple, compared to inclusion polymorphism: since data types do not contain their operations as routine fields, they do not have to be restructured to contain new operations. However, generalizability is weak, because any attempt to generalize a routine by adding type parameter or removing operation parameters will change its interface, and hence require changes in all of its clients.

Compared to inclusion polymorphism, then, type parameterization provides type matching, generality, and incrementality, but not generalizability or convenience when passing operations along with data.

F-Bounded Quantification

F-bounded quantification [8, 14] combines subtyping with universal quantification. In this approach, type parameterization has the form $\Lambda t \subseteq \tau_1 \cdot e$, and produces values of type $\forall t \subseteq \tau_1 \cdot \tau_2$. Any type argument passed to t must be a subtype of the type bound τ_1, which is a type expression that may involve t. (F-bounded quantification is a generalization of bounded quantification [11], where the type bound must not involve the type parameter.) This is parametric polymorphism, despite the use of subtyping, because the type parameter of a polymorphic definition determines the interface of the definition.

F-bounded quantification allows the definition of a version of `min` with a precise, statically checkable interface.

```
min:∀T⊆⟨lt:T→Boolean⟩·T → T → T
    = ΛT⊆⟨lt:T→Boolean⟩·λp:T·λq:T·
        if p.lt(q) then p else q

type String = ⟨... lt: String → Boolean ...⟩
s1: String = ...
s2: String = ...
s3: String = min String s1 s2
```

The call to `min` is legal because `String` is a subtype of ⟨`lt:String →Boolean`⟩, the type bound. Inside `min`, `p.lt(q)` is legal because the type bound guarantees that `T` defines a `lt` taking a `T` argument. The result of the call to `min` has type `String`.

F-bounded quantification can pass operations to polymorphic routines by including them as fields of records, as is done with `lt` above. Consequently, passing operations is simpler than in simple universal quantification, but F-bounded quantification shares record subtyping's difficulties in incrementally extending data types.

Recall that inclusion polymorphism languages typically use the type checking rule

$$\frac{a : \tau_1 \qquad \tau_1 \subseteq \tau_2}{a : \tau_2}$$

F-bounded quantification provides polymorphism even without this rule, because the exact type of any data argument can be passed as a type argument. A language without this type checking rule provides type matching, as is shown by `min` above, but can only provide polymorphic data by encoding it in a polymorphic routine, as described in the previous section.

However, a language might use the subtype-based type-checking rule, and provide both inclusion polymorphism and parametric polymorphism. This would provide the ability to define polymorphic data, but would weaken the properties of type matching, just as it did in record subtype languages: in "`min String s1 s2`", the types of `s1` and `s2` might be different subtypes of ⟨`lt:String` →`Boolean`⟩, and the result could have any subtype of `String`.

Operation Inference

The discussion of universal quantification above showed that explicit type arguments and operation arguments are, at best, verbose, and at worst, an impediment to programming. For this reason, parametric polymorphism languages generally provide some form of argument inference, which infers the values of some arguments from the types of other arguments.

Cormack's and Wright's language, ForceOne [66], provides overloading polymorphism. The standard operators are associated with special overloaded routine identifiers: the "less than" operator is associated with the identifier "`'<'`", and the expression "`a < b`" is treated as the routine call "`'<'[a, b]`". ForceOne also provides parametric polymorphism: routines can declare type parameters by preceding a parameter's type with a question mark. It also provides *implicit parameters*, which follow a vertical bar in a parameter list. Calls to polymorphic routines do not specify arguments corresponding to type parameters or implicit parameters. The compiler infers type arguments from the types of the ordinary arguments. For each implicit parameter, the compiler looks for a declaration with the same name and type (after substitution of type arguments for type parameters) in the scope of the call, and passes the declared item as the argument. As an example, consider a ForceOne implementation of the "minimum" notion.

```
min: [x,y:?T | '<':[T, T] routine T ] routine T ==
    { if [x < y] x else y }
```

Here `min[3, 5]` is a legal call, with the integer type inferred for `T` and the integer comparison routine inferred for `'<'`, and returns the integer 3. Note that this form of parametric polymorphism depends heavily on overloading: `min` can be applied to any type, so long as "<" has been overloaded for it.

The implicit parameter mechanism bears some resemblance to dynamic scoping, since the identifier "<" is bound to a value with that identifier in the (dynamic) calling environment, instead of the static environment of the routine body. It has the great advantage, compared to normal dynamically scoped systems, of being statically checked. However, it does not provide true dynamic scoping, because the inferred argument is found in the static environment of the calling environment, not its dynamic environment.

Cormack and Wright described an inference algorithm for a ForceOne-like language in [16]. In that language, universally quantified types can not be passed as type arguments, but Bumbulis has extended the inference algorithm to handle that case [6]. The algorithm infers type arguments and implicit arguments. It also adapts the degree of polymorphism of arguments, if necessary. A polymorphic routine can be specialized by binding a type to a type parameter or an implicit argument to an implicit parameter. The inference algorithm automatically specializes routines if some implicit parameter requires a less polymorphic version of the routine.

The process of implicit parameter binding does not necessarily terminate, since the argument bound to an implicit parameter may itself have implicit parameters that need to be bound through specialization. However, each iteration of this process corresponds to a parameter that would have to be explicitly specified otherwise. Hence non-termination corresponds to an infinitely large program in a language without argument inference, and long inference chains correspond to very complex programs. Cormack and Wright simply place a limit on the number of iterations that their algorithm is allowed to perform, and declare a program to be invalid if inference can not be completed within that limit.

Cormack and Wright point out many advantages that operation inference has over other polymorphism mechanisms. Like all of the other parametric polymorphism mechanisms, it provides type matching. It allows very general interface definitions, since implicit parameters state exactly what operations a polymorphic routine requires. Operations are implemented as free-standing routines, not as fields of record types, so new operations can be implemented and old operations can be reimplemented easily; this provides incrementality. Argument inference provides generalizability: a routine can be replaced by a version that has a different number of implicit parameters or type parameters without affecting clients. If type parameters are added, the corresponding type arguments will be inferred automatically, and if implicit parameters are added, the new version of the routine can be accompanied by declarations that will serve as default values for the implicit arguments.

ForceOne-style languages can have simple, efficient implementations, such as the one devised

by Cormack and Wright [15]. In their scheme a type value is represented by an integer that encodes the size and alignment of the type. Since equality comparisons between types are not provided, there is no need to try to ensure that the representations of different types are distinct. A type parameter is passed like an integer parameter. A local variable whose type is not statically known is implemented as a pointer to space that is allocated on the stack upon block entry and (implicitly) deallocated upon block exit. A parameter whose type is not statically known is implemented as a pointer to space allocated at the end of the parameter list. When a routine's result type is not statically known, the returned value must always be returned on the stack, instead of in a register. Performance suffers, especially on RISC machines, whenever data is not stored in registers.

(In many cases, a ForceOne compiler could implement a call to a polymorphic routine by in-line substitution of the routine body at the call site. It could also instantiate the routine for any given argument type or implicit argument by substituting the argument into the routine body. These approaches do not work in general, because in some cases the values of inferred arguments can not be known statically.

```
f: [ a_t:?T, n:int] routine void ==  {
    temp: record[ x:T ] == ...
    if [n > 0] f[temp, n - 1];
    }
... f['x', random[]] ...
```

Here, at each recursive call to `f` the type argument inferred for `T` is a more deeply nested record type, so the number of instantiations can not be determined until run-time.)

The main drawback of operation inference is that it does not provide any way to group together meaningful collections of types and implicit parameters. A set of type identifiers and implicit parameters that are defined for those types defines a notion, much as an abstract superclass does, and an abstraction mechanism would be a useful facility.

Descriptive Classes

Sandberg defined the language X2[1][46, 47] in a deliberate attempt to provide the flexibility of dynamically typed-checked object oriented programming languages in a statically typed language.

[1] also known as X-2

X2's "classes" are record types, or record type generators with type parameters. Sandberg gives as an example a homogeneous list class, where the element type is a parameter.

```
class list(T) is
    first: T;
    tail: list(T);
end class;
```

The type parameter allows stronger type checking of instances of `list` than does inheritance from an abstract superclass, which is the equivalent operation in object oriented languages. However, there is no notion of a subtype relationship between X2 classes, and X2 classes can not inherit from other classes. They do not contain "methods"— ordinary routines are used instead—so classes can not provide encapsulation; a separate module system restricts access to identifiers. In other words, X2 classes have little in common with the "classes" of object-oriented languages, and I will call them "types" or "type generators" instead.

Polymorphic routines use simple type parameters, called *free types*, in their interfaces. The values of the corresponding type arguments for a call are statically inferred from the calling context. Given an "`is_empty`" routine that returns `true` if a list is empty, a routine that returns the last element of a list, regardless of the list's element type, would be written and called as follows.

```
procedure last(a:list(T)) returns T;
where T is free
begin
    loop
        if is_empty(a.tail) then
            return(a.first);
        end if
        a := a.tail;
    end loop;
end last;

var a: list(int);
    b: list(real);
    i: int;
```

```
    r: real;
begin ...
    i := last(a);            -- T inferred to be int
    r := last(b);            -- T inferred to be real
```

(Note that the type parameter `T` can be used by itself or as an argument of a type generator.) This provides parametric polymorphism equivalent to simple type parameters.

To avoid having to pass operations on type parameters as explicit routine parameters, Sandberg introduced *descriptive classes* [46]. Simple descriptive classes have a type parameter, and specify routine interfaces in terms of the type parameter. As an example, Sandberg defines a descriptive class `order` that is one possible description of the notion of "totally ordered type".

```
descriptive class order(T) is
    eq: proc(order(T), order(T)) returns bool;
    grt: proc(order(T), order(T)) returns bool;
    less: proc(order(T), order(T)) returns bool;
end class;
```

An `instance` statement states that a type satisfies a descriptive class by stating which routines correspond to the routine interfaces declared by the descriptive class. For example, an instance statement declares that `int` satisfies `order` by naming the three (built-in) routines that correspond to the routine interfaces declared in `order`.

```
instance of order(int) is equal, greaterthan, lessthan;
```

Polymorphic routines can use descriptive classes in their interfaces, for example as parameter types, with free types used as arguments of the descriptive classes. If a parameter's type is a descriptive class, the corresponding argument in a call can be of any type that satisfies the descriptive class. Routines declared by the descriptive class can be used in the routine body. The `min` example could be written

```
procedure min(p:order(T); q:order(T))
returns order(T);
where T is free
begin
    if p.less(p, q) then
```

```
            return p;
        else
            return q;
        end if
    end min;

    var i, j, k: int;
    begin ...
        k := min(i, j);    -- valid: int satisfies order(T)
```

The presence of `T` in both parameters and the return type forces all three to have the *same* type, thus providing type matching. Furthermore, X2 guarantees that `min(i,j)` and `min(j,i)` have the same value, because the `instance` statement above stated that `lessthan` is the `less` operator for all `int`s used as ordered values.

Descriptive classes are quite unlike types. For instance, they can not be instantiated. They can also define *relationships* between two types:

```
descriptive class collection(c, e) is
    add: proc(e, c);
    remove: proc(e, c);
    ...
end class;
```

This descriptive class declares routines that must exist if type `c` is a collection of elements of type `e`. Hence, `collection` does not correspond to the element type or the collection type.

However, Sandberg treats descriptive classes as types. The `min` routine treats `p` as if it has a field named `less`, which was declared in `order`. This suggests that `p`'s type is `order(T)`, not `T`. Sandberg gives other examples that use `order(T)` as the type of a local variable and as an argument of a type generator. "Relationship" descriptive classes like `collection` are treated as types by arbitrarily equating the descriptive class with the first type argument. A routine might have a parameter of type `collection(f, g)` (where `f` and `g` are free), and programs might contain statements like `instance of collection(list(int), int)` to allow lists of integers to be passed to that routine; within the routine, `collection(c, e)` would refer to `list(int)`.

Another difference between descriptive classes and types is that instance statements can define if-and-only-if relationships among types. Sandberg describes a descriptive class `pane`, for panes in

a window system, and a type generator `merge(f,g)`, which is a type of pane with two sub-panes of types `f` and `g`. A single instance statement can declare that any instance of `merge` is a `pane` if and only if its two argument types are `pane`s. (Presumably, `int` is not a `pane`, and hence `merge(int, int)` would be a legal type but would not be a `pane`.) This is an intriguing capability, but it is hard to see how it can be used. If the arguments of `merge` are not panes, then how can routines with `merge` parameters make use of them? Note that this is not a hypothetical example: it is taken from a windowing system written in X2.

Descriptive classes resemble F-bounded quantification in that `collection(c, e)` resembles the type expression

```
type Collection[c, e] = ⟨add:e→c × remove:e→c × ...⟩
```

An instance statement establishes a relationship that resembles subtyping. A routine to count the elements in a collection has similar types in either system:

```
procedure count(p:collection(c, e)) returns int
where c, e are free
begin
    ...
end count;

count = Λe·Λc ⊆ Collection[c, e]·λp:c· ...
```

Descriptive classes are like abstract superclasses in that they define interfaces but not implementations. An instance statement corresponds to inheritance from an abstract superclass. A polymorphic routine with descriptive class parameters can call routines declared by descriptive classes, which provides dynamic binding, and the polymorphic routine can be applied to any type that satisfies the descriptive class, which is similar to inheritance of methods defined in abstract superclasses. Types can satisfy many descriptive classes, which is like using multiple inheritance, but without the problem of resolving conflicts among inherited definitions (but only because no class variables or methods are inherited).

Sandberg argues that descriptive classes and parameterized types lack many of the drawbacks of class hierarchies. There is no fixed class hierarchy, just instance statements that relate types and descriptive classes within a scope, so different parts of a program can easily set up different

relationships. Descriptive classes can be defined independently of the types that satisfy them (and can have smaller scopes); subclasses can not be defined before their superclasses.

Descriptive classes have two other points in their favor. First, programmers can extend types with new routines at any point, so X2 has better incrementality than record subtype languages. Second, operations declared in descriptive classes are naturally covariant instead of contravariant.

However, descriptive classes provide only a two-level hierarchy: types, and descriptive classes. Descriptive classes can not be arranged in "subclass" hierarchies: there is no way to state that a `numeric` descriptive class is like the `ordered` class, with extra arithmetic operations. Instead, logically-related descriptive classes must contain identical copies of declarations, instead of sharing them. Similarly, types that satisfy a class do not automatically satisfy classes at logically "higher" levels, so programs must contain multiple instance statements to define all of the relationships that are needed.

Another difference is that descriptive classes do not provide any support for type definition via inheritance. This is a restriction on reusability, but a minor one, since programmers can simulate inheritance by declaring a field of the "parent" type inside the child.

Sandberg leaves a number of interesting questions undiscussed.

- What are X2's restrictions on type parameter inference? Are either of these legal?

```
procedure p(i:int) returns T where T is free
procedure p(i:int) returns order(T) where T is free
```

- Does X2 support polymorphic data? The method described on page 31 depends on the ability to define routine values that refer to non-local data; X2 may not provide that facility.
- A program may need a version of a polymorphic routine that has been specialized for a particular type, for instance for use as a procedure argument or in an instance statement. Must a programmer write this specialization, or can specialization happen implicitly?
- Can the parameter of a type generator be a descriptive class? This seems obviously desirable, but if it is possible, then surely `merge` would be declared to take only `pane` parameters, and then Sandberg would not have composed his "if-and-only-if" example.

Parameterized Modules

Many programming languages provide some form of *module*: a programming language construct that contains definitions of routines, types, variables, constants, etc. This provides grouping and

also controls the "name space" of a program, since entities with the same name that are defined by different modules can be distinguished easily. Examples of modules include the Mesa *program module* [37], the Ada *package*, the SML *structure* [36, 35], and the OBJ2 *object* [25, 21]:

```
obj PAIR is
    extending INT, BOOL.
    op x: -> Int.
    op y: -> Int.
    op lt: Int Int -> Bool.
    vars i1, i2: Int.
    eq: lt(i1, i2) = (i1 < i2).
    eq: x = 5.
    eq: y = 7.
endo
```

The module `PAIR` uses two other modules, `INT` and `BOOL`, which define the types `Int` and `Bool`. The lines beginning with "`op`" define integer values named `x` and `y` and a routine named `lt`. The `vars` line declares variables used in the definitions of `lt`. The first "`eq`" line declares that, for any two integers `i1` and `i2`, `lt(i1, i2)` has the value `i1 < i2`. The other `eq` lines define the values of `x` and `y`.

The other portions of a program use modules indirectly via *module interfaces*, which declare the facilities that the modules make available. The items that are defined by modules but not declared by any interface are inaccessible outside of their module; hence interfaces provide encapsulation. An interface is called a *definition module* in Mesa, a *package specification* in Ada, a *signature* in SML, and a *theory* in OBJ2:

```
th LT_PAIR is
    extending BOOL.
    sort T.
    op x: -> T.
    op y: -> T.
    op lt: T T -> Bool.
    vars t1, t2: T.
    eq: lt(t1, t2) = not lt(t2, t1).
endth
```

This interface declares a type (a *sort* in OBJ2's terminology) called `T`, two values of type `T` called `x` and `y`, an `lt` routine for type `T`, and a logical condition that `lt` must satisfy. When `Int` is used for `T`, `PAIR` matches this interface.

Ada, SML, and OBJ2 also provide parameterized modules[2] Ada *generic packages* can have data, routine, and type arguments. SML *functors* and OBJ2 parameterized objects take modules as arguments, and use module interfaces as their "parameter types". The operation of *instantiation* substitutes arguments for the parameters to produce an ordinary module. Instantiation is a static process, not part of the execution of the program; all arguments must be constant.

Parameterized modules provide polymorphism at two levels. At the level of the module's contents, a parameter of a module defines a common structure that all arguments must have, and the contents of the module are well-defined for any such argument. In effect, the parameters of the module become parameters of the contents, and the contents exhibit parametric polymorphism. Instantiation specializes the contents. At this level of polymorphism, the polymorphic minimum routine would be placed in a parameterized module.

```
obj MIN[O::LT_PAIR] is
    op min: T.O T.O -> T.O.
    vars t1, t2: T.O.
    eq: min(t1, t2) = if lt.O(t1, t2) then t1 else t2 fi.
endo
```

(OBJ2's "dot" notation reverses the usual convention: `T.O` is the sort `T` defined in parameter `O`.) This differs from universal quantification in that it is not part of the type system. The `min` routine does not have a type, and in fact can not be used directly; a program that uses `min` must first instantiate `MIN` with an argument module to produce a monomorphic routine, which can be applied to data values.

The other form of polymorphism occurs at the higher, module, level: an SML functor or an OBJ2 parameterized object is a function from objects to objects, and instantiation plays the role of a routine call. Polymorphism at this level is used to build large programs by composing them from smaller modules. The rules for matching modules and interfaces provide an inclusion polymorphism mechanism that resembles record subtyping, and any object that matches the parameter interface is a valid argument. In SML, *curtailment* allows a structure to match a

[2]Most object-oriented languages use classes provide name space control, grouping, and encapsulation, and hence do not support modules. Eiffel extends this approach and provides parameterized classes

signature that lacks some of the structure's contents or that has weaker relationships between them. In OBJ2, a *view* does the same, and also renames the object's contents to correspond to a theory's contents names as well.

```
th T is
    sorts U, V.
    op x: -> U.
    op y: -> V.
endth

view PAIR_T of PAIR as T is
    sorts U to Int, V to Int.
endv
```

The view `PAIR_T` describes how module `PAIR` can match interface `T`: use `x` and `y`, ignore `lt`, and use `Int` for `U` and `V`. This view of `PAIR` weakens the relationship between the types of `x` and `y`, and hides `lt`. OJB2 also provides "default views" in cases where the correspondence between a theory and an object is sufficiently obvious: the match between `PAIR` and `LT_PAIR` does not require an explicit `view` statement.

Module-level polymorphism can be used to define a "minimum" function on the contents of an `LT_PAIR` object:

```
obj MIN_OBJ[O::LT_PAIR] is
    op m: -> T.O.
    eq m = if lt.O(x.O, y.O) then x.O else y.O fi.
endo
```

Recall that record subtyping weakens type matching. Similarly, module views weaken "theory matching". This is important because, in programs built from layers of modules, it may be necessary that middle-level modules share the same lower-level modules: a compiler would normally require that its scanner module and parser module use the same token type. Without "theory matching", this can not be done. Both SML and OBJ2 have extra facilities to solve this problem. SML provides a *sharing declaration*, which states that two modules, or two types defined in module arguments, are the same. A compiler module written in SML would declare that its scanner and parser arguments share their token types. OBJ2 provides *parameterized theories* for the same purpose.

```
th TOKEN is
    sort Token.
endth

th SCANNER[T::TOKEN] is ... endth

th PARSER[T::TOKEN] is ... endth

obj COMPILER[T::TOKEN, S::SCANNER[T], P::PARSER[T]] is ... endo
```

`COMPILER` takes as its first argument a module that matches `TOKEN` and substitutes it into `SCANNER` and `PARSER` to produce the theories that `S` and `P` must match. Hence the scanner and parser arguments must use the same `Token` sort. `COMPILER` exhibits parametric polymorphism at the module level instead of at the module-contents level.

(SML provides a third level of polymorphism, independent of the module system. A routine that returns the smaller of two data arguments can be written

```
fun min lt x y = if lt(x,y) then x else y;
```

This declaration omits type information, but the routine has a static type none the less. SML systems infer the type from the body of the routine; in this case, `min` has a type parameter called `'a`, and has type `('a →'a →bool) →'a →'a →'a`, for any type `'a`. SML systems also infer type arguments: in the expression "`min (op <) 5 7`", inference gives `'a` the value `int`. ("`op <`" returns the "less than" operation for integers.) Mitchell and Harper [39] show that, at this level, SML can be treated as an abbreviated form of an explicitly typed parametrically polymorphic language, similar to Girard's and Reynold's lambda calculus, but with the restriction that routines can not have polymorphic arguments; hence I will not discuss this level of polymorphism any further.)

The greatest weakness of parameterized modules is that the polymorphism that they provide is strictly a static phenomenon. There are no executable module-valued expressions: the "module expressions" of SML and OBJ2 build new modules from old ones prior to program execution. All entities involved in executable expressions are monomorphic, perhaps produced by instantiation. SML provides no sharing declarations for values or routines, since it can not statically check the equality of dynamically-computed values. Since there is no dynamic rebinding of module

identifiers, modules do not provide polymorphic data, even though views can hide information about the types of a module's constants.

Given this limitation, parameterized modules still provide useful polymorphism. They allow strong static checking of types at the module-contents level and of module interfaces at the module level. They are efficient, because entities involved in executable expressions are monomorphic. They separate interfaces and implementations at the contents and the module levels. Type matching exists at the contents level, and although it does not exist at the module level, SML and OBJ2 provide other facilities to fill the gap. At the contents level, parameterization allows much the same language simplifications as was the case for universal polymorphism. Parameterized modules provide high generality, since they can define exactly the types and operations that they need. They are somewhat generalizable, since a parameterized module can replace an existing parameter interface with one that contains fewer declarations or has weaker relationships between declarations, but it can not increase its generality by adding new declarations without affecting clients. Finally, they lack incrementality, just as records do: incrementally extending a module to match some theory would require building a new structure with extra declarations.

2.3 Summary

Polymorphism is the ability to implement a notion so that it applies to more than one type. Two main categories of polymorphism mechanisms exist. *Ad-hoc polymorphism* is present when an implementation has one or more interfaces that are defined for a set of types which need not have any common structure. *Universal polymorphism* is present when an implementation has a single interface that is applicable to a potentially infinite set of types defined by a common structure. Universal polymorphism in turn has two subcategories. A routine exhibits *parametric polymorphism* when the value of a parameter defines other parts of the routine's interface. *Inclusion polymorphism* is present when an identifier can be bound to values from a potentially infinite set of types with a common structure.

Three distinct mechanisms can provide *ad-hoc* polymorphism in a programming language. Overloading lets programmers define several implementations of a notion, and chooses among them based on the context of each use of the notion. Transfer functions implicitly convert an argument in a routine call to the type of the routine's parameter. Set-theoretic unions are defined in terms of member types, and can take on any value from those member types. All three are reasonably efficient and allow strong static checking and separation of interfaces from implementations.

All three provide, at most, modest increases in generality, generalizability, and incrementality. Overloading and set-theoretic unions allow simplification of programming languages, by allowing the language to present certain notions as library facilities instead of as special language constructs. Overloading allows precise interface definitions because it provides type matching in interfaces. Expressiveness increases, because one name can be used for many implementations of a notion. This lets programmers write algorithms in terms of those notions, instead of distinct implementations of the notions.

However, programmer-defined *ad-hoc* polymorphism provides only modest improvements, because the set of applicable types must be explicitly specified, and hence is finite. (This is not true for language-defined polymorphism. A language can implement a notion for an infinite set of types. Consider assignment, which works on an infinite set of "assignable" types. For most languages the common structure of these types (if there is any) is not expressible in the language, so *ad-hoc* polymorphism is present.) Hence, implementations of notions that use these mechanisms will not necessarily be usable with types defined afterwards, even if the notion is appropriate for those types. (`DOUBLE1` can not be applied to a new `COMPLEX` type, even though fractions can be doubled.)

Two mechanisms provide inclusion polymorphism. Infinite unions provide a type that contains all values of all other types; the generality of this type means that almost nothing can be done with instances of it. Hence any operations that must be performed on values passed to a parametric routine must be passed along with the value, as separate parameters or as fields of a data structure. Record subtyping defines subtype relationships among record types and among routine types, and allows routine fields of records to refer to the record through a special identifier such as `self`. The record fields conveniently package values and operations on the values. This approach provides strong static checking, separation of implementations from interfaces, generality and generalizability, simpler base languages, and polymorphic data, with reasonable efficiency. However, it provides only weak incrementality, and does not allow type matching. The subtype rule for routines is contravariant, and this limits expressiveness. Languages based on record subtyping often combine code reuse, encapsulation, and subtyping in the "class" construct, which further limits expressiveness and flexibility.

Parametric polymorphism also provides strong static checking, simple base languages, and separation of interfaces from implementations with reasonable efficiency. The simplest form, universal quantification, allows routines to have type parameters. This provides generality, incrementality, type matching, and polymorphic data (through an awkward encoding), but again it becomes awk-

ward to pass operations along with the values they operate on. F-bounded quantification specifies a type expression that type arguments must be subtypes of: this directly provides polymorphic data, and allows programmers to package values and their operations, in the manner of record subtyping, but with the advantage that the operations can exhibit covariance. However, this packaging reduces incrementality and flexibility, and subtyping reduces type matching. Both forms of type quantification lack generalizability. Automatic inference of type arguments and their associated operations provides generalizability, while preserving the properties of incrementality and type matching. Compared to the inclusion polymorphism languages, however, parametric polymorphism with argument inference lacks a mechanism for abstracting over types and operations. The X2 language provides such a mechanism, in its descriptive classes; however, X2 attempts to combine descriptive classes with types, and does not allow for hierarchies of abstractions. A parameterized module is another such mechanism, but modules are completely static entities, and (like the records that they resemble) they limit generalizability and incrementality.

Chapter 3

Contextual Polymorphism

This chapter proposes a new polymorphism mechanism, *contextual polymorphism*, that addresses the weaknesses of the mechanisms described in the previous chapter. It is based on parametric polymorphism, and therefore provides precise types, generality, incrementality and generalizability. It extends parametric polymorphism with contexts, which represent notions within the programming language.

A context is an abstraction of a collection of declarations. Typically the declarations are related in some way; they might declare operations and constants associated with some notion. Contexts have type parameters, and if the type generators of the programming language can take arguments whose types are ordinary data types, then contexts can have parameters of those types; for instance, in a language with array types indexed by integers, contexts can have integer parameters. Any form of declaration provided by the language might be allowed in the body of a context; routine and object declarations are most useful, but there is no reason to exclude type declarations.

An assertion produces a collection of declarations, typically by applying contexts to arguments. It asserts that the declared identifiers exist and have the declared types and kinds. Typically assertions are used to declare that some type implements a notion.

3.1 Definition of Contexts and Assertions

The sections that follow give a formal explanation of contexts and assertions in terms of $F_\omega^\exists$, an explicitly-typed polymorphic lambda calculus based on F_ω, which is a variant of Girard's infinite-

Kinds k, k_i	$=$	$*$	(the kind of term types)
	$\mid$	$\forall t : k_1 \ni \Gamma \cdot k_2$	(the kind of type generators)
Type variables t, t_i			
Types τ, τ_i	$=$	t	(simple types)
	$\mid$	$\tau_1 \rightarrow \tau_2$	(function types)
	$\mid$	$\forall t : k \ni \Gamma \cdot \tau$	(universally quantified types)
	$\mid$	$\lambda t : k \cdot \tau$	(type generators)
	$\mid$	$\tau_1 \tau_2$	(type generator application)
	$\mid$	(τ)	
Environments Γ, Γ_i	$=$	$\emptyset$	(the empty environment)
	$\mid$	$x : \tau$	(term variable declarations)
	$\mid$	$t : k$	(type variable declarations)
	$\mid$	$\chi t : k \cdot \Gamma$	(contexts)
	$\mid$	$\Gamma \tau$	(context application)
	$\mid$	Γ_1, Γ_2	(environment concatenation)
	$\mid$	(Γ)	
Term variables x, x_i			
Terms e, e_i	$=$	x	
	$\mid$	$\lambda x : \tau \cdot e$	(functions)
	$\mid$	$e_1 e_2$	(function application)
	$\mid$	$\Lambda t : k \ni \Gamma \cdot e$	(polymorphic functions)
	$\mid$	$e[\tau]$	(type application)
	$\mid$	(e)	

Figure 3.1: The abstract syntax of $F_\omega^\ni$

order polymorphic lambda calculus [42]. The abstract syntax of $F_\omega^\ni$ is given in figure 3.1. Each rule of the syntax defines a grammar symbol, some syntactic variables that represent instances of the symbol, and the productions for that symbol (if it has any). The unusual features of the syntax are the two symbols $\ni$, which introduces assertions and should be read as "such that", and χ, which introduces contexts.

The calculus has a number of rules for static checking. An expression that passes these checks is evaluated by repeated application of two β-reduction rules, for function application and type application:

$$(\lambda x : \tau \cdot e_1)e_2 \quad \longrightarrow_\beta \quad e_1[x := e_2]$$
$$(\Lambda t : k \ni \Gamma \cdot e)[\tau] \quad \longrightarrow_{\beta'} \quad e[t := \tau]$$

where $l[m := n]$ denotes the substitution of n for all free occurrences of m in l, with renaming of variables in l to avoid the capture of variables that are free in n. β reduction corresponds to execution of ordinary programs.

3.1.1 Types and Kinds

$F_\omega^\ni$ associates types with every value. Besides simple types, function types, and universally quantified types, $F_\omega^\ni$ has *type generators*, which are functions from types to types. The programming language might provide pre-defined simple types such as `Integer` and predefined type generators such as `List`, which would take a type argument and return the type of homogeneous lists of the argument type. "`List Integer`" is an example of type generator application, and has as its value the type of lists of integers.

$F_\omega^\ni$ also associates a *kind* with every type. Kinds provide information about the parameters of type generators, and allow checking of type generator application in the same way that types allow checking of function application. The kind "$*$" is the kind of the term types (types with no parameters): every term has a type, and that type has kind $*$. For instance, the term `3` has type `Integer`, which has kind $*$. Type generators take type arguments, so their kinds involve "$\forall$"[1], just as polymorphic routines take type arguments and have types that involve "$\forall$". `List` has kind $\forall t : * \ni \emptyset \cdot *$, since it takes a term type as an argument and returns a term type. (The significance of "$\ni\emptyset$" is explained below.)

[1] In most lambda calculi, type generator kinds use $\rightarrow$ instead of $\forall$, in analogy with function types; `List`'s kind would be $*\rightarrow *$. The $\forall$ notation used here, inspired by [38], provides a name for the type parameter, and the assertion can use that name.

3.1.2 Environments, Contexts, and Assertions

An *environment* Γ relates term variables to their types, and type variables to their kinds. The simplest environment is $\emptyset$, the empty environment, which contains no information. Other environments contain declarations. A *declaration* has the form $t : k$ (which declares that type variable t has kind k), or $x : \tau$ where $\tau : *$ (which declares that term variable x has type τ). The domain of an environment, $\mathrm{dom}(\Gamma)$, is the set of type and term variables it declares.

The operation of *environment concatenation* combines two environments that have no variables in common.

$$\Gamma_1, \Gamma_2 = \Gamma_2 \cup \Gamma_2 \quad \text{iff } \mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset$$

Concatenation associates left to right: $\Gamma_1, \Gamma_2, \Gamma_3 \equiv (\Gamma_1, \Gamma_2), \Gamma_3$. Since the two environments declare distinct variables, certain absurdities caused by redeclaration of type variables can not occur. For instance, "$(t : *, x : t), t : \forall u : *_\ni \emptyset \cdot *$" is undefined, so the question of x's type in that environment does not arise. (It might be possible to allow redeclaration of term variables while forbidding redeclaration of type variables, but treating all variables consistently provides simplicity.)

Despite this restriction on environment concatenation, lambda expressions may redeclare variables, because the type and kind checking rules allow consistent renaming of bound variables: the expression $\lambda x : \tau_1 \cdot \lambda x : \tau_2 \cdot x$ is legal because $\lambda y : \tau_1 \cdot \lambda x : \tau_2 \cdot x$ is.

Contexts and assertions are the most complex environment components, and are the distinguishing features of $F^{\ni}_{\omega}$. A *context* $\chi t : k \cdot \Gamma$ abstracts from the declarations in the environment Γ, which may make use of the type parameter t[2]. Normally, the parameters of contexts have kind "$*$", but higher kinds are allowed; in those cases, the argument must be a type generator instead of a type.

An *assertion* $_\ni\Gamma$ produces a collection of declarations, called *assertion declarations*, typically by applying a context to a type. Polymorphic routines and type generators (and their types and kinds) use them to constrain their arguments. The trivial assertion "$_\ni\emptyset$" produces no declarations. The type generator `List` has kind $\forall t : *_\ni \emptyset \cdot *$ because it takes a term type argument, places no constraints on it, and produces a term type.

Contexts and context application play no part in the evaluation of $F^{\ni}_{\omega}$ expressions: like ordinary environments, they are part of the static type-checking rules of the language. The "value"

[2]Since type generators in $F^{\ni}_{\omega}$ only take types as arguments, contexts only have type parameters. A formal explanation of contexts in languages where type generators also take type arguments might be based on the Calculus of Constructions [41] instead of $F\omega$.

of a context application $(\chi t : k \cdot \Gamma)\tau$ is the substitution $\Gamma[t := \tau]$. Repeated application of this substitution converts environments that contain context applications to normal forms that contain only concatenated declarations:

$$\begin{array}{lcll} NF(\Gamma_1, \Gamma_2) & = & NF(\Gamma_1), NF(\Gamma_2) & \\ NF((\chi t : k \cdot \Gamma_1)\tau) & = & NF(\Gamma_1[t := \tau]) & \\ NF(\Gamma) & = & \Gamma & \text{otherwise} \end{array}$$

The following environment contains a context that abstracts some declarations associated with the notion "boolean type", and uses it to produce declarations of those operations on the type `Bit`.

$$\text{Bit} : *, (\chi t : * \cdot \text{and} : t{\to}t{\to}t, \text{or} : t{\to}t{\to}t, \text{not} : t{\to}t, \text{true} : t, \text{false} : t)\text{Bit}$$

For the sake of readability, most examples below use the *context statement*

```
context x = Γ in e
```

which means "$e[x := \Gamma]$".

```
context boolean = χt:*·and: t → t → t, or: t → t → t,
not: t → t, true: t, false: t
in ... ΛBit:* ∋ boolean Bit· ...
```

3.1.3 Judgements

Environments are used to type-check $F_\omega^\ni$ programs. A *type judgement* has the form $\Gamma \vdash e : \tau$, and means that e has type τ given the environment Γ. Similarly, the *kind judgement* $\Gamma \vdash t : k$ means that type t has kind k given Γ.

Environments used in type and kind judgements must meet certain conditions: variables must be declared only once, the type of every term variable must have a kind, and every context must be applied to an argument of the right kind. Therefore $F_\omega^\ni$ needs a mechanism for checking the kinds of context arguments, just as types and kinds allow for the checking of arguments of functions and type generators. The mechanism is the *context sort*.

$$\begin{array}{rcll} \text{Context sorts } \gamma, \gamma_i & = & \Box & \text{(the sort of environments)} \\ & | & k{\to}\gamma & \text{(the sort of contexts)} \end{array}$$

An environment that may be used in judgements has sort $\Box$. By definition, $\emptyset$ has sort $\Box$. A context whose parameter is a type of kind k has the sort $k{\to}\gamma$.

The *sort judgement* $\Gamma_1 \vdash \Gamma_2 :: \gamma$ states that, given the environment Γ_1 with sort $\Box$, environment Γ_2 has the context sort γ.

The first two sort checking rules state that "$t : k$" is an environment, and that "$x : \tau$" is an environment if τ has kind $*$.

$$\frac{}{\emptyset \vdash t : k :: \Box}$$

$$\frac{\Gamma_1 \vdash \tau : *}{\Gamma_1 \vdash x : \tau :: \Box}$$

The next two rules deal with context formation and application. They state that a type argument of a context must have the same kind as the context's parameter, and that the parameter is part of the environment of the context's body.

$$\frac{\Gamma_1 \vdash \tau : k \qquad \Gamma_1 \vdash \Gamma_2 :: k \rightarrow \gamma}{\Gamma_1 \vdash \Gamma_2 \tau :: \gamma}$$

$$\frac{\Gamma_1, t : k \vdash \Gamma_2 :: \gamma}{\Gamma_1 \vdash \chi t : k \cdot \Gamma_2 :: k \rightarrow \gamma}$$

The final rule applies to environment concatenation. Again, the concatenated environments must declare distinct variables.

$$\frac{\Gamma_1 \vdash \Gamma_2 :: \Box \qquad \mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset}{\emptyset \vdash \Gamma_1, \Gamma_2 :: \Box}$$

3.1.4 Type Judgements

This section gives the type checking rules for terms. Recall that the environment Γ used in a type judgement $\Gamma \vdash e : t$ must have sort $\Box$.

The first rule states that a term variable has the type given by the environment.

$$\frac{\emptyset \vdash \Gamma :: \Box}{\Gamma \vdash x : \tau} \qquad \text{if } x : \tau \in NF(\Gamma)$$

The next rules define the types of functions and function applications. They state that the type of a function's argument must match the parameter type, and that the parameter declaration is part of the environment of the function's body.

$$\frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1 e_2) : \tau_1}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x : \tau_1 \cdot e) : \tau_1 \rightarrow \tau_2}$$

Polymorphic functions $\Lambda t : k \ni \Gamma \cdot e$ use assertions to constrain the type (or type generator) variable t by declaring identifiers that use it in their types. The type checking rule for polymorphic functions adds the assertion declarations to the environment of the function bodies.

$$\frac{\Gamma_1, t : k, \Gamma_2 \vdash e : \tau}{\Gamma_1 \vdash (\Lambda t : k \ni \Gamma_2 \cdot e) : \forall t : k \ni \Gamma_2 \cdot \tau}$$

(Note that, by the definition of concatenation, $\Gamma_1, t : k, \Gamma_2 \vdash e : \tau$ requires that $\Gamma_1, t : k \vdash \Gamma_2 :: \Box$ and that Γ_1, Γ_2 and $t : k$ have no declarations in common.)

The rule for calls to polymorphic functions checks that the type argument has the same kind as the function's parameter, substitutes the type argument for the function's parameter in the assertion declarations, and checks that the resulting declarations exist in the environment of the call.

$$\frac{\Gamma_1 \vdash \tau_1 : k \qquad \Gamma_1 \vdash e : \forall t : k \ni \Gamma_2 \cdot \tau_2 \qquad \Gamma_2[t := \tau_1] \subseteq \Gamma_1}{\Gamma_1 \vdash e[\tau_1] : \tau_2[t := \tau_1]}$$

When a polymorphic function applies a context to its type argument, and the calling environment contains the assertion declarations, the type argument is said to *satisfy* the assertion.

Most examples below use the *let statement* "`let` $x : \tau$ `=` e_1 `in` e_2", which means "$(\lambda x : \tau \cdot e_2)(e_1)$". Since τ is always e_1's type, many examples omit it when it is irrelevant, cumbersome, or obvious from inspection of e_1.

(The definition of the `let` statement uses function application, because the expression e_1 can be calculated when the program is evaluated. In contrast, the definition of the `context` statement uses substitution, because contexts are strictly compile-time phenomena and hence vanish before function application occurs.)

Consider this example, which defines a polymorphic routine that doubles its argument.

```
context addable = χT:*·plus:T → T → T in
let plus = λx:Integer·λy:Integer·(x + y) in
let double = ΛT:* ∋ addable T·λx:T·plus x x
in double[Integer] 5
```

Static checking requires the substitution of `addable` and application to its argument.

```
let plus = λx:Integer·λy:Integer·(x + y) in
let double = ΛT:* ∋ plus:T→T→T·λx:T·plus x x
in double[Integer] 5
```

A `plus` routine for `Integer` exists in the environment of the call to `double`, so `Integer` is a legal argument for `double`. β-reduction of `double` and the first argument gives

```
let plus = λx:Integer·λy:Integer·(x + y)
in (λx:Integer·plus x x) 5
```

Note that, after this β-reduction, the use of `plus` in the body of the polymorphic routine refers to the `plus` defined in the calling environment. The assertion has the effect of connecting the routine's body to the calling environment.

Since substitution renames variables to avoid capture of free identifiers, the declarations in a context are unaffected by declarations in inner scopes. For instance, the perverse example

```
context c = χT:*·f:T → Integer
in (ΛInteger:*·ΛT:* ∋ c T·e)[Boolean]
```

redefines `Integer`. However, it is equivalent to

```
(Λt_1:*·ΛT:* ∋ f:T → Integer·e)[Boolean]
```

and hence the redefinition does not affect the use of `Integer` in `c`. In contrast, if contexts were simply expanded like macros, the redefinition would have affected the use of `c`.

3.1.5 Kind Judgements

This section gives the kind checking rules for types. Again, the environments used by kind judgements must have sort $\Box$.

The first three rules state that type variables have the kind given by the environment, and that function types and universally quantified types are term types.

$$\frac{\emptyset \vdash \Gamma :: \Box}{\Gamma \vdash t : k} \quad \text{if } t : k \in NF(\Gamma)$$

$$\frac{\Gamma \vdash \tau_1 : * \qquad \Gamma \vdash \tau_2 : *}{\Gamma \vdash (\tau_1 \rightarrow \tau_2) : *}$$

$$\frac{\Gamma_1, t : k, \Gamma_2 \vdash \tau : *}{\Gamma_1 \vdash (\forall t : k \ni \Gamma_2 \cdot \tau) : *}$$

Type generators use assertions to constrain their arguments, just as polymorphic functions use assertions to constrain their type arguments. The kind checking rules for type generators require

that the argument of a type generator have the same kind as the type generator's parameter, and that the environment of a type generator application contains the assertion declarations, much as was the case for type applications.

$$\frac{\Gamma_1, t : k_1, \Gamma_2 \vdash \tau : k_2}{\Gamma_1 \vdash (\lambda t : k_1 \ni \Gamma_2 \cdot \tau) : \forall t : k_1 \ni \Gamma_2 \cdot k_2}$$

$$\frac{\Gamma_1 \vdash \tau_1 : k_1 \qquad \Gamma_1 \vdash \tau_2 : \forall t : k_1 \ni \Gamma_2 \cdot k_2 \qquad \Gamma_2[t := \tau_1] \subseteq \Gamma_1}{\Gamma_1 \vdash (\tau_2\, \tau_1) : k_2}$$

The *type statement* "`type t =` τ `in` e" is a synonym for $e[t := \tau]$. It does not provide an abstract type facility, because it substitutes τ into e before the type rules can be applied, and so all implementation details are visible to e.

```
type SortedList: ∀T:* ∋ lt: T → T → Boolean·*
= λT:* ∋ lt: T → T → Boolean·List T in
let  primes: SortedList Integer = e1
in e2
```

The assertion declares that the argument of `SortedList` must be a type with a `lt` function, so the environment of the declaration of `primes` must contain a declaration of `lt : Integer→Integer→Boolean`. This constraint on type arguments resembles the "property lists" of parameters of the `form` type generator in Alphard [48, p. 38], or the `where` clause of a parameterized Clu cluster [33].

Note that the declaration of `primes` does not bind the current value of `lt` into `primes`.

```
let insert = ΛT:* ∋ lt: T→T→Boolean·λv:T·λl:SortedList T·
... (lt v (head l)) ... in
let lt = λx:Integer·λy:Integer·(gt x y) in
in  ... insert[Integer] 7 primes ...
```

Because of the rules of function application, the `lt` routine that is called by `insert` is the one visible at the call site, which is actually an alias for `gt`, not the one visible at the declaration of `primes`.

3.1.6 Specialization

The type-checking rules given above require that for each $x : \tau$ in an assertion, there be an $x : \tau$ in the environment of the call, and that the two types match exactly (after the substitution of

type arguments). It may happen that the environment contains an x that does not have type τ, but does have a polymorphic type, and that binding some of its type parameters can create a specialization to τ. For instance,

```
let f = ΛT:* ∋ g:T → T·λp:T·e1 in
let g: ∀S:*∋∅·S → S
= ΛS:*∋∅·λx:S·e2
in f[Integer] 5
```

does not type-check because the g visible at the call site does not have type `Integer →Integer`: instead, it is polymorphic. A programmer who wishes to use g must first specialize it for the type `Integer`.

```
let f = ΛT:* ∋ g:T → T·λp:T·e1 in
let g: ∀S:*∋∅·S → S = ΛS:*∋∅·λx:S·e2 in
let g: Integer → Integer = g[Integer]
in f[Integer] 5
```

(The call to g in the inner g is not a recursive call; `g[Integer]` is evaluated outside of the scope of the inner declaration of g, by the definition of `let`.) If specialization must remove some type parameter other than the first, a slightly more complex expression will suffice.

```
let h: ∀T1:*∋∅·∀T2:*∋∅·T1 → T2
= ΛT1:*∋∅·ΛT2:*∋∅·e1 in
let h: ∀T1:*∋∅·T1 → Integer
= ΛT1:*∋∅·h[T1][Integer]
in e2
```

Specialization can remove assertion declarations as well as type parameters. In this example the inner g removes x from the outer g's assertion by binding in a call to the polymorphic x. The result can be passed to f, which requires an argument that has no assertion declarations.

```
let f = λg:∀T:*∋∅·T → T·e1 in
let g: ∀T:* ∋ x:T → T·T → T
= ΛT:* ∋ x:T → T·λy:T·e2 in
let x = ΛT:* ∋ ∅·T → T.e3 in
let g = ΛT:*∋∅·(let x:T → T = x[T] in g[T])
in f g
```

This situation is a modest improvement over the corresponding situation in programming languages with plain universal quantification, since in those languages the specialization must appear as an explicit parameter. However, the necessity of creating the specialization still hampers generalizability, since generalizing a routine may force many clients to create specializations. (Furthermore, since the routine does not appear as an explicit argument, locating these clients will be difficult.)

3.1.7 Argument Inference

$F_\omega^\ni$ as defined above requires that type arguments of polymorphic functions appear explicitly, and it requires explicit definition of specializations. It also does not allow overloading of term variables. All three of these restrictions can be removed by applying an algorithm like the one for ForceOne-style languages described by Cormack and Wright and Bumbulis, which was mentioned in section 2.2.2. Programs written in $F_\omega^\ni$ can be converted to the desired form by rewriting their assertion declarations as implicit parameters, and then the Cormack-Wright-Bumbulis algorithm can be applied to the result.

Once overloading is allowed, the definition of environment concatenation reduces to simple set union. If two environments being concatenated declare the same term variable with the same type (or the same type variable with the same kind), the union operation combines the declarations. If they declare the same variable with different types (or kinds), the union operation preserves both declarations, and overload resolution distinguishes between them.

Type argument inference emphasizes the difference between Λ-binding and λ-binding of type parameters: arguments to Λ-bound parameters may be inferred, but arguments to λ-bound parameters must always be specified explicitly.

3.2 Use of Contexts and Assertions

3.2.1 Guidance

A programming system that provides contexts will, over time, build up a library of contexts that represent useful notions. These contexts can guide programmers who wish to implement other, related notions. This guidance has three aspects.

- It specifies the complete set of operations that are necessary to implement the notion.

- It gives the required names and types of the operations. This information is crucial in a polymorphism system based on overloading.
- It gives information about programming style. Consider the notion "stack of integers", and a possible implementation in the C programming language that uses linked lists.

```
struct element {
    int             i;     /* A value on the stack */
    struct element *next; /* Next stack element   */
};
typedef struct element* IntStack;
```

The push operation might be implemented in a "functional" style, where it returns a new stack with a new value on top.

```
IntStack push(IntStack s, int v) {
    IntStack newelt = malloc(sizeof(struct element));
    newelt->i = v;
    newelt->next = s;
    return newelt;
}
```

It might also use an "object-oriented" style, where it modifies an existing stack and returns nothing.

```
void push(IntStack *sp, int v) {
    IntStack newelt = malloc(sizeof(struct element));
    newelt->i = v;
    newelt->next = *sp;
    *sp = newelt;
    return;
}
```

These styles produce different contexts, since the types of the operations differ. If programmers stick to the style described by available contexts when they implement related notions, more opportunities for reuse will occur.

3.2.2 Abstract Superclasses

Contexts resemble pure abstract superclasses in that they define interfaces of instances of types, not implementations of types. An assertion that a type satisfies a context serves many of the same purposes as a declaration that a class inherits from an abstract superclass. However, type declarations do not explicitly list the contexts that they satisfy, so new contexts can be added to describe existing types, and new operations can be added to allow existing types to satisfy existing contexts. Also, contexts do not force operation types to follow a contravariant pattern. Given the context

```
context addable = χT:*·plus:T → T → T
```

the assertions `addable Integer` and `addable Fraction` would produce the covariant declarations

```
plus:Integer → Integer → Integer
plus:Fraction → Fraction → Fraction
```

Contravariant routine subtype rules force abstract superclass-based system to use imprecise declarations:

```
Addable = ⟨... plus:Addable → Addable ...⟩
Integer = ⟨... plus:Addable → Integer ...⟩
Fraction = ⟨... plus:Addable → Fraction ...⟩
```

The subtype-style declarations also allow mixed-mode operations, with their benefits and difficulties. The `addable` context does not; mixed-mode addition operations would require a more complex, multi-parameter context.

Contexts and polymorphic routines can be used in the manner of impure abstract superclasses to help programmers to provide full implementations of notions. Consider `collection`, which declares useful operations and values for collection types `C` with element type `E`.

```
context collection = χC:*·χE:*·
nil: C,                    -- An empty collection.
is_empty: C → Boolean,     -- Is the collection empty?
add: C → E → C,            -- Add an element to a collection.
```

```
drop: C → C,              -- Remove front from the collection.
front: C → E,             -- Return an arbitrary element.
map: C → (E → E) → C      -- Apply a function to every element.
```

The map operation can be implemented using the other operations. The basic operations can be abstracted into a second context, collection_base, and a polymorphic map routine can use collection_base.

```
context collection_base = χC:*·χE:*·
nil: C,
is_empty: C → Boolean,
add: C → E → C,
front: C → E,
drop: C → C
in let map = ΛC:*∋∅·ΛE:* ∋ collection_base C E·
λc:C·λf:E → E·e1
in e2
```

A programmer implementing a new collection type would only have to write the routines named by collection_base; the type would satisfy collection because the polymorphic map can be specialized for it. (The programmer might still choose to write a type-specific version of map for efficiency reasons.)

In a class-based language, a programmer can use multiple inheritance from several abstract superclasses as a way of defining a class that has several interfaces. Each interface provides a different view of instances of the class, for use by different clients. The equivalent situation in $F^{\ni}_{\omega}$ is a type that simultaneously satisfies many contexts. A buffer type could satisfy "producer" and "consumer" contexts; a library type could have different views for clients that use the type directly, and for clients that extend the type to create new types. In the case of the class, the set of interfaces is limited by the set of superclasses in the class's declaration, while in $F^{\ni}_{\omega}$ an existing type can satisfy newer contexts. Furthermore, when multiple inheritance is used, the problem of multiple inherited implementations of a single entity arises. This problem does not occur with contexts, because contexts only contain declarations; there is no need to select or combine implementations.

Finally, the collection context above describes a relationship between two types: "instances of type C contain elements of type E". It does not correspond to any single abstract superclass;

instead it corresponds to a family of abstract superclasses with distinct element types. An object-oriented language with parameterized classes could imitate `collection` by declaring a parameterized abstract superclass for collections (corresponding to `collection`'s `C` parameter) that takes an element type (corresponding to the `E` parameter) as an argument. However, contexts can describe relationships among types that parameterized classes can not express in a natural way. One example is a "transitive conversion" relationship.

```
context tcvt = χA:*·χB:*·χC:*· cvt: A → B, cvt: B → C in
let cvt = ΛA:*∋∅·ΛB:*∋∅·ΛC:*∋ tcvt A B C·
            λa:A·cvt (cvt a)
```

"Position" and "offset" types provide a second example. A position value marks a place along a dimension. An offset value represents the distance between two positions. Calendar dates are position values, where the offsets represents a number of days. Similarly, memory addresses are position values, where the offsets represent numbers of bytes. Only certain operations between positions and offsets are meaningful.

```
context dimension = χPosition:*·χOffset:*·
    +: Position → Offset → Position,
    +: Offset → Position → Position,
    -: Position → Offset → Position,
    -: Position → Position → Offset in
... dimension Date Day ...
... dimension Address Integer ...
```

3.2.3 Context Hierarchies

Since assertions are environments, they can sensibly appear in the bodies of contexts. For example, `collection` can be rewritten as

```
context collection = χC:*·χE:*·
    collection_base C E,
    map: C → (E → E) → C
```

This produces the same set of declarations as the original version of `collection`. (It produces a set, not some sort of nested structure, because the "," concatenation operation produces the union of its operands.)

The use of one context in another creates hierarchies of contexts, which reflect relationships between the corresponding notions. This resembles the use of inheritance in object-oriented languages to create hierarchies of abstract superclasses. The "child" context contains (and hence reuses) the declarations in the "parent" contexts. Multiple inheritance corresponds to the use of more than one assertion, but once again, the problem of multiple inherited implementations of an entity does not arise.

3.2.4 Abstract Types

Recall that the type statement does not provide an abstract type facility. However, within the body of a polymorphic function $\Lambda t : *_\exists \emptyset \cdot e$, the type t appears to be a primitive type, even though the type argument passed to it need not be. Reynolds [44] used this as the basis of an abstract type facility. A program that uses an abstract type is treated as a polymorphic routine, parameterized by the abstract type. Operations on the type are treated as function parameters of the program. The implementation of the abstract type and the routines that implement its operations are passed as arguments to the program. The program

```
abstype Complex with plus:Complex → Complex → Complex
in  let c1: Complex = ... in
    let c2: Complex = ... in
    ... plus c1 c2 ...
is  ⟨re:Real × im:Real⟩
    with plus = λx: Complex·λy: Complex·e1
```

would be modeled in $F^{\exists}_{\omega}$ by

```
(ΛComplex:*∃∅·λplus:Complex → Complex → Complex·
   let c1: Complex = ... in
   let c2: Complex = ... in
   ... plus c1 c2 ...)
[⟨re:Real × im:Real⟩]
(λx:⟨re:Real × im:Real⟩·λy:⟨re:Real × im:Real⟩·e1)
```

Assertions allow a modified version of this scheme. The abstract type is still represented as a type parameter, but its operations are declared by an assertion, which defines operations that can be performed on the abstract type; it is similar to an `import` statement.

```
abstype Complex ∋ addable Complex
in  let c1: Complex = ... in
    let c2: Complex = ... in
    ... plus c1 c2 ...
is  ⟨re:Real × im:Real⟩
    with plus = λx: Complex·λy: Complex·e1
```

The translation of this sort of program into $F^{\ni}_{\omega}$ is more complex, because the type parameter must be passed to the program within a scope where the assertion declarations are defined. It would be modeled by

```
let plus = λx: ⟨re:Real × im:Real⟩·
λy: ⟨re:Real × im:Real⟩·e1
in  (ΛComplex:* ∋ addable Complex·
       let c1: Complex = ... in
       let c2: Complex = ... in
       ... plus c1 c2 ...)
    [⟨re:Real × im:Real⟩]
```

3.2.5 Existential Types

Mitchell and Plotkin [40] describe a connection between abstract types and existential types, which have the form $\exists t \cdot \tau$. The intuitive meaning of $x : \exists t \cdot \tau$ is that some unknown type with kind $*$ exists such that, if it were named t, then x would contain a value of type τ. Mitchell and Plotkin treat t as an abstract type, and use a *tuple* (or unlabeled record) type for τ; then the tuple fields are operations on (and values of) the type. A *data algebra* is a value that has an existential type. It contains implementations of the abstract type and the values and operations in τ. A `pack` expression creates data algebras out of implementations of the abstract type and the tuple fields. A variant of `abstype` explodes data algebras and gives a name to the type and to each tuple field. Using these facilities, the `complex` example used above becomes

```
abstype Complex
    with plus:Complex → Complex → Complex
in  let c1: Complex = ... in
    let c2: Complex = ... in
```

```
        ... plus c1 c2 ...
    is  pack ⟨Real × Real⟩,
             λx: ⟨Real × Real⟩·λy: ⟨Real × Real⟩·e1
        to ∃t·⟨t → t → t⟩
```

Data algebras are first class values that can be passed as parameters or returned from functions. In the example above, the pack expression could be replaced by an expression that returns either a polar or a Cartesian implementation of complex numbers, depending upon circumstances.

Universally quantified types can encode existential types, using the technique mentioned on page 31:

```
let p: ∃t·τ1
     = pack τ2, e to ∃t·τ1
```

becomes

```
let p: ∀r:*∋∅·(∀t:*∋∅·τ1 → r) → r
     = Λr:*∋∅·λo:(∀t:*∋∅·τ1 → r)·o[τ2] e
```

The abstype statement has a more complex equivalent. If the result of the statement

```
abstype a with x:τ2 in e1 is e2
```

has type τ_1 and e_2 has type $\exists t \cdot \tau_2$, the equivalent is

```
e2[τ1] (Λa:*∋∅·λv:τ2·e1)
```

except that any reference to x in e_1 must be replaced by an expression that extracts the correct field from v.

Given this equivalence, it seems natural to use a slightly different form of existential type that uses assertions.

$$\begin{aligned} \exists t : k \ni \Gamma \cdot \tau_1 &\equiv \forall r : *\ni\emptyset \cdot (\forall t : k \ni \Gamma \cdot \tau_1 \to r) \to r \\ \text{pack } \tau_2, e_1 \text{ to } \exists t : k \ni \Gamma \cdot \tau_1 &\equiv \Lambda r : *\ni\emptyset \cdot \lambda o : (\forall t : k \ni \Gamma \cdot \tau_1 \to r) \cdot o[\tau_2] e_1 \\ \text{abstype } t : k \ni \Gamma \text{ with } x : \tau_1 \text{ in } e_3 \text{ is } e_2 &\equiv e_2[\tau_3](\Lambda t : k \ni \Gamma \cdot \lambda x : \tau_1 \cdot e_3) \end{aligned}$$

where τ_1 is the type of the contents of the data algebra, τ_2 is the hidden type, and τ_3 is the result type of the abstype statement. The assertion declarations produced by Γ declare operations on

and values of the unknown type t, just as some of the fields of a tuple type do in the traditional existential type. τ_1 is made up of the remaining fields from the tuple; it can be viewed as a value held inside the data algebra. For instance, a value of type $\exists$`t:*`$\ni$`addable t`$\cdot$`t` contains a value with a type that is unknown, but is known to have a `plus` routine.

The type and kind checking rules for existential types and data algebras follow from those for universal types and polymorphic functions, and in fact are the same as those given by Mitchell and Plotkin if the parts that deal with contexts, assertions, and kinds are ignored. An existential type is a term type, because the equivalent universal type is.

$$\frac{\Gamma_1, t : k, \Gamma_2 \vdash \tau_1 : *}{\Gamma_1 \vdash (\exists t : k \ni \Gamma_2 \cdot \tau_1) : *}$$

The type-checking rule for the `pack` statement follows from the rules for the equivalent Λ expression. The environment of a `pack` statement must contain the assertion declarations. This is analogous to the way that assertion declarations are checked against the calling environment when a type is passed to a polymorphic function.

$$\frac{\Gamma_1 \vdash \tau_2 : k \qquad \Gamma_1 \vdash e_1 : \tau_1[t := \tau_2] \qquad \Gamma_2[t := \tau_2] \subseteq \Gamma_1}{\Gamma_1 \vdash (\text{pack } \tau_2, e_1 \text{ to } \exists t : k \ni \Gamma_2 \cdot \tau_1) : \exists t : k \ni \Gamma_2 \cdot \tau_1}$$

Similarly, the type rule for `abstype` follows from the rules for the equivalent type application. The `abstype` statement has an assertion that adds the assertion declarations to the environment of its body, just as the assertion in a polymorphic function adds to the environment of the function body. It also declares a name for the value held in the data algebra.

$$\frac{\Gamma_1 \vdash e_2 : \exists t : k \ni \Gamma_2 \cdot \tau_1 \qquad \Gamma_1, t : k, \Gamma_2, x : \tau_1 \vdash e_3 : \tau_3}{\Gamma_1 \vdash (\text{abstype } t : k \ni \Gamma_2 \text{ with } x : \tau_1 \text{ in } e_3 \text{ is } e_2) : \tau_3}$$

(As in [40], t must not be free in τ_1 or in the type of any variable other than x in e_1, to ensure that the representation of the abstract type does not escape from the `abstype` statement.) The semantics of the modified `pack` and `abstype` statements are that the values bound to the assertion declarations in the environment of `pack` are implicitly packaged in the data algebra, and made available in the body of `abstype`. In Mitchell and Plotkin's original version, those values had to be explicitly packed and unpacked.

Existential types with assertions are not fundamentally different from ordinary existential types, but they encourage a different interpretation. $\exists t \cdot \langle \ldots \rangle$ can be thought of as the type of (or the interface of) a module; the fields of the tuple type are the operations and values declared by the module, and a data algebra with that existential type implements the module. The separation of assertion declarations from the "data" portion of $\exists t : * \ni \Gamma \cdot t$ suggests treating it as an unknown

type t that is known to support the operations and constants in Γ; it defines a polymorphic *data* type (whereas $\forall t : *\ni\Gamma \cdot \tau$ defines a polymorphic routine type). Between these two views lie existential types with non-trivial assertions and with non-trivial bodies that reveal some structure: given the declaration

```
x:∃T:* ∋ addable T·⟨T × T⟩
```

we do not know the type of the fields of x, but we know that they have exactly the same type, and that they can be added together.

A data algebra with a trivial body, $\exists t : *\ni\Gamma \cdot t$, has an unknown type but a known set of operations, and therefore resembles an object in an object-oriented language. This can be combined with other type generators. For instance, `List ∃t:*∋addable t·t` is a polymorphic list type, where every element has an `addable` type, but different elements can have different actual types.

Programs that compute with data algebras tend to unpack, manipulate, and repack them without making any mention of the hidden type. For instance, values of an existential `addable` type can be doubled by the routine

```
let double: (∃T:* ∋ addable T·T) → (∃T:* ∋ addable T·T)
    = λx:(∃T:* ∋ addable T·T)·
        abstype s with y: T in
            pack s, (plus y y) to ∃T:* ∋ addable T·T
        is x
```

The verbosity of this example suggests that an abbreviation would be useful. The following `open` statement rebinds the name of a data algebra so that it refers to the *contents* of the algebra and repacks the result of an expression using the contents.

$$\frac{\Gamma_1 \vdash x : \exists t : k \ni \Gamma_2 \cdot \tau_1 \qquad \Gamma_1, t : k, \Gamma_2, x : \tau_1 \vdash e : \tau_2}{\Gamma_2 \vdash (\text{open } x \text{ in } e) : \exists t : k \ni \Gamma_2 \cdot \tau_2}$$

```
let double: (∃T ∋ addable T·T) → (∃T:* ∋ addable T·T)
    = λx:(∃T:* ∋ addable T·T)·
        open x in (plus x x)
```

3.2.6 Inheritance

Assertions, when used with type declarations, provide a facility similar to subtyping. The "class" construct of object-oriented languages provides a second facility: inheritance of routines and fields. Assertions can provide some support for inheritance-like facilities by using them with type definitions.

Classes can be simulated by record types, and field inheritance by having the "child" record type contain an instance of the "parent". Routine inheritance can be simulated by writing a "cover" routine for the child type that extracts the parent field and passes it to the inherited routine.

```
type Region = ⟨origin:Point⟩ in
let move: Region → Point → Region
        = λr:Region·λp:Point·... in
type Rect = ⟨parent:Region × bottom_right:Point⟩
let move = λr:Rect·λp:Point·move r.parent p
```

This can be automated by interpreting an assertion in the scope of a type *definition* to be a declaration of routines that must be defined in the scope of the definition. If the programmer leaves some of the routines undefined in that scope, then the language must construct cover routines from similar routines that exist for its fields type.

```
context region = χT:*·move:T → Point → T in
type Region = ⟨origin:Point⟩ in
let move = λr:Region·λp:Point·... in
type Rect ∋ region Rect = ⟨parent:Region × bottom_right:Point⟩
```

In this case, the `move` routine for `Rect` would be created automatically, because the assertion `region Rect` requires that it exist. This is akin to the default definition of certain common operations (such as assignment, or equality comparisons), to the way that Ada defines derived routines for derived types [57, §3.4], and to Wirth's record extensions [64].

3.3 Contexts and Polymorphism

Notions such as "collection" are useful because of the operations that can be performed on them. Types that implement some version of the notion will implement the operations with routines,

and corresponding routines will have the same name, and will have types that share a pattern of argument and result types. The declarations of these routines can be abstracted into a context. In that case, the context is a programming-language representation of the notion, and an assertion that a type satisfies the context also asserts that the type implements the notion. Hierarchical connections among contexts reflect connections among the corresponding notions.

Contextual polymorphism is a form of parametric polymorphism, where assertions define the structure that arguments are required to have. It allows precise, general definitions of interfaces, since assertions can describe exactly what operations a polymorphic routine needs, and because the use of named type parameters provides type matching. An automatic type argument inference and specialization mechanism, similar to that of ForceOne, provides generalizability and incrementality. Contexts do nothing to hinder the separation of routines and routine interfaces, and provide a way to specify type interfaces separately from types, by specifying operations supported by the type. Among its drawbacks are a lack of support for data and routine inheritance, and a lack of direct support for polymorphic data. Polymorphic data can be provided indirectly, through the encoding discussed in section 2.2.2, or through some other facility, such as an existential type mechanism.

3.3.1 Subtypes and F-Bounded Quantification

Contextual polymorphism does not involve any notion of subtyping. However, assertions are related to the concept of "subtyping". If a context has one type parameter, then all of the types that satisfy it are in some sense subtypes of an imaginary type containing all of the values of those types. If this imaginary type is identified with the context, then an assertion that a type satisfies the context resembles a declaration that the type is a subtype of the imaginary type. However, this analogy breaks down for contexts with more than one parameter. Contexts can express relationships that are more complex than "is a subtype of".

Contexts and assertions resemble F-bounded quantification's type functions and type bound expressions: compare

```
context ordered = χT:*·lt:T → T → Boolean in
let min = ΛT:* ∋ ordered T·λx:T·λy:T·
    if (lt x y) then x else y
in e
```

and

```
Ordered[T] = ⟨lt: T → Boolean⟩
min = ΛT ⊆ Ordered[T]·λx:T·λy:T·
    if (x.lt y) then x else y
```

Both mechanisms constrain the types that are acceptable as arguments to the polymorphic routines, both allow static type checking by defining a set of available operations and values for the type, and both provide dynamic binding. The types of the declared variables in the two cases are similar, considering that a record is an implicit parameter of its routine fields; generally, the record type corresponds to the type parameter of the context.

However, assertions are not record types; they constrain argument types by defining declarations that must exist in the environment, instead of fields that a record must have. Hence the declared variables are directly available through normal binding mechanisms, instead of through field selection from a data parameter or a value that can be computed from a data parameter. Consider the meaning of a `plus` operation on lists of integers. It might concatenate its arguments, or it might perform pairwise additions on the list elements.

```
let l: List Integer in
let plus: List Integer → List Integer
        = ... (concatenate arguments) ...
in double l -- concatenates  l to itself.

let plus: List Integer → List Integer
        = ... (add elements together)
in double l -- doubles each element.
```

In F-bounded quantification, `plus` must be a field of the `List` type. The implementor of `List` must choose one implementation for `plus`, and either choice will be wrong for some client of `List`. In fact, using contextual polymorphism, a list could be concatenated to itself in one part of a program and doubled element-wise in another. An equivalent program written using F-bounded quantification would have to define a new type with a rebound `plus`, convert the list to the new type (so that the new `plus` operation will be available), and convert the result back to the original type. (The conversion process becomes even more complicated when an operation on the *elements* of a data structure must be rebound.) This difference means that there is no direct translation of a contextual polymorphism program into an F-bounded quantification program, even though assertions can be translated into record types fairly easily, because data arguments may require

extensive repackaging. It also means that contextual polymorphism has an advantage over F-bounded quantification in flexibility and incrementality, because programs can more easily rebind or add operations to meet local needs.

F-bounded quantification can not specify interfaces as precisely as contextual polymorphism, because subtyping weakens the property of type matching. Contextual polymorphism may have a small advantage in efficiency because of this, and also because entities declared by assertion declarations are directly accessible, while the fields of a record type are only available indirectly via field selection.

3.3.2 Parameterized Modules

Contexts resemble OBJ2's theories and SML's signatures. The declarations in a context's body correspond to the "op" declarations in a theory, and the type parameter of a context corresponds to a `sort` declaration in a theory. Applying a context in a routine corresponds to using a theory in a parameterized object that defines the routine. The type-checking rules that look for assertion declarations in the calling environment corresponds to the use of a default view, and an explicit `view` corresponds to a set of declarations that rename values. As an example, the `ordered` context given above corresponds to the theory

```
th ORDERED is
    extending BOOL.
    sort T.
    op lt: T T -> Bool.
endth

obj MIN[O::ORDERED] is
    op min: T.O T.O -> T.O.
    vars t1, t2: T.O.
    eq: min(t1, t2) = if lt.O(t1, t2) then t1 else t2 fi.
endo

view INT_ORDERED of INT as ORDERED is
    sort T to Int.
    vars i, j: T.
```

```
    op: lt(i, j) to i < j.
endv
```

(At first glance, it seems that context parameters should correspond to theory parameters instead of sort declarations, but that approach is needlessly complex. If `ORDERED` took a parameter and defined `lt` in terms of the parameter's sort, then `MIN` would need two parameters: one called `T` to provide the sort, and the other (matching `ORDERED[T]`) to provide `lt`.)

Both contexts and theories can describe relationships between types. However, a theory describes the interface of a single argument module, while a context can be applied to several type arguments. This means that OBJ2 programs must sometimes construct modules solely to match theories. Consider how the two systems would implement a routine to sum the elements of a collection.

```
context sum_collection = χC:*·χE:*·
    plus: E → E → E,
    front: C → E,           -- Return an arbitrary element.
    ...
in let sum: ∀C:* ∋ ∅·∀E:* ∋ sum_collection C E·C → E = ...
```

```
th SUM_COLLECTION is
    sorts C E.
    op plus: E -> E -> E.
    op front: C → E.
    ...
endth
obj SUM[SC::SUM_COLLECTION] is
    op sum: C.SC -> E.SC.
    ...
endo
```

Now consider an OBJ2 program with a `POINT` module that provides a `Point` type, a `POINTSET` module that provides a `PointSet` type, and a `POINTPLUS` module that provides a `plus` operation on points. To sum a set of points, an OBJ2 program must create a new module that combines `POINTSET` and `POINTPLUS` and can be viewed as a `SUM_COLLECTION`. A contextual polymorphism program would just pass `Point` and `PointSet` to `sum`.

The main difference between contexts and theories is the level that they operate on. Theories operate at the level of modules, and are compile-time phenomena. Contexts operate at the level of types, and although they also are statically checked, the types that they constrain are dynamic values. The routines that use them can be called through statically-unpredictable paths, or can even be recursive; OBJ2 and SML do not allow recursive modules.

3.3.3 Descriptive Classes

Contexts closely resemble X2's descriptive classes. Compare the `ordered` context above to

```
descriptive class ordered(T) is
    lt: proc(ordered(T), ordered(T)) returns ordered(T);
end class;
procedure min(x:ordered(T), y:ordered(T)) returns ordered(T);
where T is free
begin
    if x.less(x, y) then
        return x;
    else
        return y;
    end if
end min;
```

Both systems allow the same "abstract superclass" style of programming, and both have the same advantages with respect to the lack of a fixed inheritance hierarchy, the ease of adding new contexts and new operations, and the avoidance of multiple inheritance's problems.

X2 uses an explicit `instance` statement, where contextual polymorphism implicitly checks assertions against the calling environment. Hence contextual polymorphism can not match the `instance` statement's ability to state if-and-only-if relationships among types. An instance statement also provides useful documentation, and provides a way to rename routines; however, the `sort_list` example of the previous section shows that renaming is simple in $F^{\ni}_{\omega}$.

The two mechanisms differ most in that assertions are not used as types; they constrain type parameters, instead of serving as parameter types. This resolves some questions left unanswered in the discussion of X2. The X2 declarations

```
procedure p(i:Integer) returns order(T) where T is free
l:list(order(T))
```

may or may not be legal, but $F_\omega^\ni$ does not present these problems because contexts and types are distinct entities. Furthermore, $F_\omega^\ni$ allows hierarchies of contexts and defines the meaning of assertions applied to the parameters of type generators. X2 does not allow hierarchies of descriptive parameters, and Sandberg does not discuss the use of descriptive types with type generators.

3.4 Summary

This chapter has described a variation of parametric polymorphism called contextual polymorphism. This form of polymorphism provides a larger role for environments within a programming language. Environments contain declarations of types and terms. Contexts are abstractions of environments, in the same way that routines are abstractions of terms. Assertions apply contexts to arguments to produce declarations. Contextual polymorphism uses assertions to constrain the type parameters of polymorphic routines and type generators; the items declared by an assertion must exist in the environment of a call to the polymorphic routine or a type generator application. Inference algorithms infer type arguments of, and generate specializations of, polymorphic routines. Assertions can be extended to other programming language mechanisms, such as abstract types and existential types.

Contextual polymorphism preserves the desirable properties of other parametric polymorphism variants: strong static checking, separation of interfaces from implementations, precise interfaces (including type matching within an interface), generality, generalizability, and incrementality. Contexts also provide an abstraction mechanism that, like the supertype relationships of inclusion polymorphism, provides hierarchical descriptions of notions, and guidance and assistance to programmers. Contexts can also describe relationships among types, such as "type `C` is a collection of elements of type `E`".

Chapter 4

Notes on Cforall

If new ideas about programming languages are to be tested, or even effectively demonstrated, they must be embedded in a language. This is often done by creating a new language. This shows the ideas off to their best advantage, and provides the language designer with all of the satisfactions that come from designing a new thing from scratch. However, new languages only rarely gain wide-spread acceptance. A second approach uses the ideas to extend an existing language. This helps people examine the new ideas, because they can bring their knowledge of the base language to bear. It also exposes the new ideas to a somewhat hostile environment, so their flaws will be more readily apparent.

This chapter discusses some features of Cforall, a "test bed" for contexts, assertions, and related ideas, that extends ANSI standard C [2]. The purpose of Cforall's design is to show that contexts and assertions provide a practical and efficient form of polymorphism, and that these ideas lead to a smaller, simpler language kernel. The choice of C as a base language is somewhat arbitrary. Many statically-typed imperative languages could serve as a base for this work, but C's traditional emphasis on efficiency is an important constraint, its history of other extensions (such as C++ [19] and Objective-C [17]) provides interesting comparisons, and the availability of source code for high-quality C compilers (such as gcc [51] and lcc [20]) will ease the job of implementation.

4.1 Overloading

Contextual polymorphism depends crucially on overloading: related notions are implemented by declarations of the same identifier with related types, and contexts abstract out the pattern common to the types. Hence Cforall must allow extensive overloading of routine and object identifiers. C's semantics restrict (and encourage) overloading in certain ways.

Two declarations of an identifier that have overlapping scopes overload each other unless

- they have compatible types, according to C's type compatibility rules.
- one declares a routine, and the other declares a pointer to a compatible routine type. (Since C technically treats routine identifiers as pointers to routines in expressions, the two cases are indistinguishable.)
- one declares an array type, the other declares a pointer type, and the element type and the pointed-at type are compatible. (Since C treats arrays as pointers in expressions, the two cases are indistinguishable. This is a restriction, since it is conceivable that a programmer might want to define an operation for pointers and for the elements of an array.)
- one declaration declares a type or typedef name. (C's grammar requires that type names be distinct from other identifiers. This is a restriction, since it would be convenient to use the same identifier for a type and for a routine that constructs values of the type.)

If one of these conditions holds, and the declarations occur in different scopes, the inner one hides the outer one; otherwise, the declarations are illegal.

Cforall programmers can overload most of C's operators by declaring routines with special identifiers; for instance, the identifier "`?+?`" represents the addition operator, and the expression "`a + b`" is considered to be syntactic sugar for the routine call "`?+?(a, b)`". (The question marks represent the routine's operands: the identifier for the unary "+" operation is "`+?`".) Programmers can not overload the "flow of control" operators "`&&`" (short-circuit "and"), "`||`" (short-circuit "or"), and "`?:`" (the conditional expression) because, unlike routines, they do not always evaluate their operands.

The constants "`0`" and "`1`" are important to the semantics of C. Besides their normal roles as arithmetic identities, "`0`" represents the null pointer and the boolean "false" constant, and plays a role in the definitions of the logical operators, while "`1`" represents the "true" constant and is involved in the definition of many arithmetic operators, including pointer arithmetic. Along with

"`!=`", they define the unoverloadable flow-of-control operators: "`&&`" returns `1` if both operands are unequal to `0`, and returns `0` otherwise. Consequently, overloading "`0`" and "`1`" is highly desirable, and Cforall allows it by treating those tokens as special identifiers.

C and Cforall use the word *object* to mean a contiguous region of a computer's data memory. Variables, arrays, structures, and structure fields are all objects, but routines are not. Unlike Ada, Cforall allows overloading of object identifiers, because the names of constants like "`0`" and "`1`" overloadable, and a convenient way to implement a constant for some type T is to declare an object of type `const T`. Unlike C++, Cforall allows an identifier to be overloaded for two routine types that differ only in their result types; it did not seem sensible to allow `T1 foo` and `T2 foo`, and then forbid `T1 bar(void)` and `T2 bar(void)`.

C considers all enumeration constants to have type `int`. Therefore, enumeration constants can not be overloaded in Cforall, and enumeration types are considered to be compatible with `int`[1]. Furthermore, C and Cforall consider all character literals to have type `int`, not `char`. These decisions restrict overloading severely. C++ does not have to follow C as closely as Cforall does, and in this situation C++ treats enumeration types as distinct types, enumeration constants as values of their enumeration types, and character literals as `char` values.

Cforall treats C's overloadable operators and "`0`" and "`1`" constants as a group of predefined routines and objects. They are treated as being defined in each translation unit, so that compilers can perform optimizations such as in-line expansion on a translation unit (as opposed to program-wide) basis. They have addresses (although the result of comparing pointers to them is implementation dependent), and can be passed as parameters to other routines.

4.2 Lvalues

The expressions of $F_\omega^\ni$ only dealt with values. C and Cforall also provide *lvalue* expressions, which designate a region of data storage. Lvalues usually occur in assignment, subscripting, and pointer dereference expressions. Contextual polymorphism must handle lvalues as well as ordinary values.

The simplest way to deal with an lvalue is to eliminate it and use a pointer value instead. For instance, Cforall allows programmers to overload assignment, using the identifiers "`?=?`" for simple assignment, "`?+=?`" for increment, and so on. The first parameter of an assignment routine points to the lvalue to be modified. The assignment expression "`a=b`" is equivalent to

[1] ANSI C requires that enumeration types be compatible with *some* integer type, but not necessarily `int`.

the routine call "`?=?(&a,b)`", where the "address-of" operator "`&`" returns a pointer to `a`. This guarantees that `a` is an lvalue, since only lvalues have addresses. However, this scheme does not handle assignments to register variables or to bit fields, since Cforall has no "pointer to register" or "pointer to bit field" types. For those sorts of lvalues, Cforall uses the programmer-defined assignment operator to assign to a temporary variable, then copies the temporary to the lvalue.

The subscript and dereference operators return lvalues, and can not be easily rewritten to use pointer values instead. These cases can be handled by an "lvalue" type generator. Cforall provides this through an `lvalue` type qualifier. For example, a programmer who created a `Flex` type that implements an expandable array of integers would want to overload the subscript operator as follows:

```
lvalue int ?[?] (Flex array, int index);
```

The `lvalue` qualifier indicates that the result of subscripting a `Flex` is an integer lvalue, not just an integer value.

4.3 Polymorphism

4.3.1 Type Abstraction

Cforall provides a `forall` specifier that provides type abstraction, like $\Lambda T \cdot \tau$ in a polymorphic lambda calculus. A polymorphic `swap` routine can be written as

```
forall(type T)
void swap(T* t1, T* t2) {
    T temp = *t1;
    *t1 = *t2;
    *t2 = temp;
}
int i1, i2;
struct { double x,y; } s1, s2;
swap(&i1, &i2);
swap(&s1, &s2);
```

Here, "`type T`" declares a parameter identifier, `T`, which denotes a type within the declaration of `swap`. The compiler infers the corresponding type argument from the data arguments in calls to `swap`. `T` takes on the values `int` and `struct { double x,y; }` in the two calls shown.

Cforall has no equivalent of the lambda calculus's $e[\tau]$ type application term. The compiler must always infer all type arguments. Cforall restricts the form of `forall` declarations to make this possible. The body of the declaration (the equivalent of τ in $\Lambda t \cdot \tau$) must have a routine type, and every type parameter must be inferable from the "ordinary" parameters: it must be used in the type of at least one ordinary parameter (or, as is discussed below, in the type of an assertion declaration whose type mentions some other inferable type parameter). Besides making inference possible, this restriction sidesteps questions about the legality of polymorphic data type declarations like

```
forall(type T) struct { T a, b; };
```

`type` is not a type specifier, so there are no such types as "pointer to `type`", "array of `type`", or "routine returning `type`", and no variables or structure members of type `type`. Cforall provides no operations on instances of `type`; in particular, types can not be assigned or compared. These restrictions allow simple, efficient implementations of polymorphic routines. Their first consequence is that a type specifier such as `T` always refers to the same type value during the lifetime of any of its instances. Hence a Cforall compiler can statically keep track of the types and type identifiers used to declare any object, and can reconstruct its type when necessary. This means that objects and values do not have to contain type tags. The second consequence is that type values can have a simple representation, such as the one described for the language ForceOne in section 2.2.2.

In contrast, `T` does act as a type specifier. It can be used in declarations: for instance, "pointer to `T`", "array of `T`", and "routine returning `T`" are legal types. It can also be used as the operand of `sizeof`. Instances of `T` can be passed as parameters, assigned to each other, initialized, and used as the operands of `sizeof` or "`&`", but no other operations for type `T` exist by default. Consequently, the value referred to by a type identifier must be an *object type* (a type that describes an object). This excludes

- `type` (which can not occur in pointer or structure declarations), bit field types (which can only occur in structure declarations), and routine types. ("Pointer to routine" types are not excluded.)

- incomplete types: `void`, unbounded array types, and structure types that have been declared, but whose members have not been declared. Fortunately, unbounded arrays types do not come up in practice, because C converts "array of type T" to "pointer to type T" in expressions. Unfortunately, this restriction makes it impossible to write a polymorphic routine that accepts a pointer to any data type, and uses the pointed-at type to relate the types of parameters or the return type. The only alternatives have the form

  ```
  forall(type T) T* f(T*, T*);
  ```

  which will not accept pointers to incomplete types, or

  ```
  void* f(void*, void*);
  ```

  which does not preserve type information.

- type qualifiers: `const` and `volatile`. In a sense, they qualify objects, not types. They can not be part of type values, because they affect the semantics of objects declared with those type values: `volatile` affects the storage and retrieval of objects, and `const` affects the set of legal operations. As a result, programmers must often overload polymorphic routines that operate on pointers for all four combinations of qualifiers, so that the result type will match the argument type.

  ```
  forall(type T) const volatile T* f(const volatile T*);
  forall(type T) volatile T*       f(volatile T*);
  forall(type T) const T*          f(const T*);
  forall(type T) T*                f(T*);
  ```

4.3.2 Contexts and Assertions

Cforall's context declaration is the equivalent of the `context` statement defined for $F_\omega^\ni$. A simple context declaration looks like

```
context addable(type T) {
    T       ?+?(T, T);
    const T 0;
};
```

A context for a notion of a homogeneous list type is

```
context list_of(type List, type Element) {
    Element car(List);
    List    cdr(List);
    List    cons(Element, List);
    List    nil;
    int     is_nil(List);
};
```

The body of a context can contain any object or routine declaration. Contexts must have at least one parameter, and all of the parameters must be types. (At present, contexts can not have data parameters because they would have little use: the only data values that appear in type expressions are the integer bounds of array types, but C treats array types as pointer types and performs no array bound checking, so the type system ignores the array bounds.)

Assertions have the form "`| list_of(Int_list, int)`". They are clauses that can be attached to any type declaration. When they are attached to type declarations in context declarations, they build context hierarchies, and provide the equivalent of the environment concatenation operation from $F^{\exists}_{\omega}$. For instance,

```
context list_of_addable(type L,
                        type E | addable(E) | list_of(L, E)) { };
```

describes the notion of homogeneous lists of some addable type, and is equivalent to

```
context list_of_addable(type List, type Element) {
    Element ?+?(Element, Element);
    const Element 0;
    Element car(List);
    List cdr(List);
    List cons(Element, List);
    List nil;
    int is_nil(List);
};
```

Assertions usually appear in `forall` specifiers.

```
forall(type L, type E | list_of_addable(L, E))
E sum(L list) {
```

```
    E total = 0;          /* E's 0, not the int 0! */
    while (!is_nil(list)) {
        total = total + car(list);
        list = cdr(list);
    }
    return total;
}

Int_list a_list;
int car(Int_list);
Int_list cdr(Int_list);
Int_list cons(int, Int_list);
Int_list nil;
int is_nil(Int_list);
...
int i = sum(a_list);
```

Given the call to `sum`, a Cforall compiler would infer from the type of `a_list` that `L` is `Int_list`. It would also find appropriately typed `car`, `cdr`, `cons`, `nil`, `is_nil`, "`0`", and "`?+?`" declarations in the environment of the call, and infer from the assertion declarations that `E` is `int`.

Cormack and Wright suggest implementing polymorphic routines by passing the routines and constants associated with the type parameters to the routine as extra, invisible arguments. The presence of contexts in Cforall suggests a somewhat different scheme; once the compiler has determined that `Int_list` and `int` satisfy `list_of_addable`, it can generate a table of the assertion declarations produced for them by `list_of_addable`. This table can be passed as a single invisible argument to `sum`, and to any other routine that uses that assertion. In many cases, the table can be created at compile time. This technique reduces the cost of calling polymorphic functions, but increases the cost of operations within the polymorphic functions. The tables resemble the "virtual function tables" created by some implementations of C++ [19, p. 227], and the "maps" created by the SELF compiler [12].

4.3.3 Local Routines

Cforall programs must sometimes contain routine definitions that are local to a scope. The simplest case occurs when a programmer must rename a routine to satisfy a context, or to temporarily

replace the default version of an operation. For example, assume that `Int_list` has an `append` routine. If the programmer renames `append` to "`?+?`", and provides a "`0`" list, `sum` can be used to concatenate all of the elements in a list of `Int_lists`. Fortunately, C and Cforall treat routines as pointers to routines; hence an initialized local pointer suffices for renaming.

```
Int_list append(Int_list, Int_list);
void foo(Int_list_list l) {
    Int_list ?+?(Int_list, Int_list) = append;
    const Int_list 0 = nil;
    ... sum(l) ...
}
```

Specialization can also create local routines, without programmer intervention. In this example, the compiler must create a specialization of `length` inside `g` in order for `Int_list` to satisfy `c`.

```
context c(type T) { int length(T); };
forall(type T | c(T)) void f(T);
forall(type L, type E | list_of(L, E)) int length(L);
forall(type L, type E | list_of(L, E)) void g(L a_list) {
    f(a_list);
}
```

Specialization is highly desirable; without it, the usefulness of polymorphic routines like `length` would be greatly reduced. However, as this example shows, the values specialized upon may be parameters of the routine body that contains the specialization. One way to implement this is to have the compiler create a local, specialized routine that passes its arguments, with extra type information, to the general routine. This implies the existence of nested routines in the style of Algol 60 or Pascal. This in turn affects the calling conventions and entry and exit code of Cforall routines.

4.4 Overload Resolution

Cforall's overload resolution rules were carefully designed so that the same rules apply to overloaded routines and overloaded operators, and so that a set of predefined "operator" routines can

provide much of C's expression semantics. The rules are based on four ideas: implicit conversions, safe conversions, conversion cost, and degrees of polymorphism.

4.4.1 Implicit Conversion

C defines a large number of implicit conversions. The rules governing conversions can vary among C compilers, because on different computers the range of values of some types may be contained in or may only overlap the range of other types, and the conversion rules attempt to preserve the converted value in the largest number of cases. The rules are:

- Values of certain "small" types (`char`, `signed char`, `unsigned char`, `short int`, and signed and unsigned `int` bit-fields) are converted to `int` or `unsigned int` values by the *integral promotions*. If an `int` can contain all values of the original type, then the result is an `int`; otherwise the result is an `unsigned int`.
- An argument of a routine call undergoes a *default argument promotion* if the type of the corresponding parameter is unknown. (This happens when the routine has not been declared with a prototype, or when the routine accepts a varying number of arguments.) The default argument promotions apply integral promotions to integral arguments and convert `float` arguments to `double`.
- Arithmetic types can all be interconverted during assignment, and during parameter passing when the type of the corresponding parameter is known.
- Pointers can be converted to and from pointers to `void`, and type qualifiers can be added to the pointed-at type.
- Values of type "array of type T" are converted to "pointer to type T" unless they are operands of `sizeof` or "`&`", or are (wide) character literals used in initializers. Routine designators are converted to pointers to routines unless they are operators of `sizeof` or "`&`". Hence, these types never actually appear in expressions, and can be ignored.
- Binary arithmetic operators make use of what are called the *usual arithmetic conversions*, in which the arguments are converted to a common type, which is the type of the result. The following ordering is defined on the arithmetic types:

  `long double` > `double` > `float` > `long unsigned int`

  `long unsigned int` > `long int` > `int`

  `long unsigned int` > `unsigned int` > `int`.

Furthermore, in implementations where `long int` can contain all of the values of type `unsigned int`,

`long int` > `unsigned int`

The first step in determining the type of the result of a binary operator in C is to apply the integral promotions to any operands that are values of a "small" type. Then, the least type that is equal to or greater than both operands' types is chosen as the result type, and the operands are converted to that type.

4.4.2 Safe Conversions

Consider defining a set of Cforall routines that mimic C's semantics as they apply to the multiplication operator. Clearly, one monomorphic routine will not do, since the result type depends on the operand types. One polymorphic routine will not do, because it would accept non-arithmetic types (unless contexts were used to restrict the set of operand types; but that leads to circular definitions, because the contexts would naturally be phrased in terms of arithmetic operations). One monomorphic routine per combination of operands is not satisfactory: the number of routines needed is large, and the exact set would be implementation dependent because of implementation-dependent conversions. Besides, the individual conversions will still be present in the language, because they can be applied to arguments of non-operator routines. It is better to define one multiplication routine for each arithmetic type, and make use of the existing implicit conversions when the operands are of different types.

In most cases, if there is a conversion from type `T1` to type `T2`, there is also a conversion from `T2` to `T1`. This leads to an ambiguity: when an `int` is multiplied by a `long`, either operand could be converted to the other's type. Cforall solves this by designating the usual arithmetic conversions, conversions to `void`, pointer conversions that add type qualifiers to the pointed-at type, and conversions of "pointer to `T`" to "pointer to `void`" to be *safe conversions*, and gives them preference during overload resolution. Other conversions are *unsafe conversions*.

Cforall must allow unsafe conversions because C allows them: a `float` argument can be passed to a routine with an `int` parameter. The usual arithmetic conversions and conversions to `void` are "safe" because they are built into the definition of C expression and statement semantics; preferring the `int`-to-`long` conversion over the reverse means that, when an `int` is multiplied by a `long`, the `int` will be converted, as is the case in C. The pointer conversions are "safe" because nothing can be done to the pointed-at value through the converted pointer that could not be

done through the original pointer; allowing these conversions reduces the number of overloadings of polymorphic routines on pointers, since all of the pointer arguments will be converted to the safest operand type.

4.4.3 Conversion Cost

Cforall mimics C's behaviour in applying the usual arithmetic conversions to binary operations by associating a *conversion cost* with each safe conversion, and by preferring overload resolutions with low costs. The following *direct safe conversions* have a cost of 1:

- from any object type or incomplete type to `void`;
- from a pointer to any non-`void` type to a pointer to `void`;
- from a pointer to any type to a pointer to a version of the type with more type qualifiers;
- an integral promotion;
- from `int` to `unsigned int`, and to `long int`;
- from `unsigned int` to `long unsigned int`;
- from `unsigned int` to `long int`, if a `long int` can represent all values of an `unsigned int`;
- from `long int` to `long unsigned int`;
- from `long unsigned int` to `float`;
- from `float` to `double`;
- from `double` to `long double`.

The cost of any other safe arithmetic conversion is the minimum number of direct safe conversions needed to produce the same result type (and hence is implementation dependent). For example, a `long int` to `long double` conversion has a cost of 4, because it corresponds to safe conversions from `long int` to `long unsigned int`, then to `float`, then `double`, and finally to `long double`.

4.4.4 Degrees of Polymorphism

Cforall defines one routine to be *less polymorphic* than another if it has fewer type parameters, or if it has the same number of type parameters and fewer of its explicit parameters have types that depend on a type parameter.

```
forall(type T, type U) void f1(T, U);     /* polymorphic */
forall(type T)         void f2(T, T);     /* less polymorphic */
forall(type T)         void f3(T, int);   /* least polymorphic */
```

Among calls that require only safe conversions, the overload resolution rules prefer calls to less polymorphic routines over calls to more polymorphic routines, because polymorphism may have some run-time cost, and because a less polymorphic routine is presumably better tuned to its arguments. However, calls to more polymorphic routines are preferred to calls to less polymorphic routines that involve unsafe conversions.

4.4.5 Overload Resolution Rules

Cforall's overload resolution rules can be explained as a bottom-up pass over an expression tree that selects an interpretation of the expression. It is similar to the overload resolution algorithm described for Ada by Baker [3], but is extended here to handle polymorphic routines and arithmetic conversions.

Each expression must have at least one *interpretation*. Each interpretation has a value and type.

- Each overloading of an identifier provides one interpretation of the identifier.
- The interpretations of expressions involving the overloadable operators are found by treating them as routine calls, as was explained in section 4.1
- Routine calls have one interpretation for each valid combination of routine interpretations and argument interpretations, where a combination is valid if the routine's interpretation accepts the number of arguments given, every argument expression can be implicitly converted to the type of the corresponding parameter, and the environment of the call contains declarations matching any assertion declarations made by the routine.

- "`s.m`" has one interpretation for each valid combination of the interpretations of `s` and `m`, where a combination is valid if `s`'s interpretation is a structure or union with a member named `m`.
- "`p->m`" has one interpretation for each valid combination of the interpretations of `p` and `m`, where a combination is valid if `p`'s interpretation is a pointer to a structure or union with a member named `m`.

An expression may have many interpretations, but the interpretations must have distinguishable types. If two or more interpretations of "`s.m`" or "`p->m`" have compatible types, then they are replaced by a single *ambiguous interpretation*. If two or more interpretations of a routine call have compatible types, the "best" of those interpretations is chosen and the rest are discarded.

- The best interpretations use the fewest unsafe conversions.
- Of these, the best are those that call the routines that are the least polymorphic.
- Of these, the best have the lowest total conversion cost, including all implicit conversions in the argument subexpressions.
- Of these, the best have the lowest total conversion cost, *excluding* the implicit conversions (if any) used to convert the argument expressions to the corresponding parameter types.

If a routine call has no single best interpretation, all of the compatible interpretations are replaced by a single ambiguous interpretation.

All interpretations of a subexpression are potential operands of higher-level expressions in the tree. Various situations select some of these interpretations.

- In a cast expression "`(t)e`", if the expression `e` has an interpretation that has type `t`, that interpretation is selected. Otherwise, `e` must have some interpretations that can be converted to type `t`, and one of them must have a conversion cost lower than any of the others, with unsafe conversions considered to have infinite cost. That interpretation must be unambiguous, and it is selected and converted.
- An expression that is used as a statement, or as the first operand of a comma expression, or as the initializing expression or incrementing expression of a `for` loop, is implicitly cast to `void`.

- An argument of a routine call is implicitly cast to the type of the corresponding routine parameter. If the type of the parameter is not known, the argument must have exactly one interpretation, which must be unambiguous, and which undergoes a default argument promotion.
- The expression in a `return` statement is implicitly cast to the return type of the routine.
- Expressions used in initializers are implicitly cast to the type of the object or aggregate element being initialized.
- An expression e used to control a loop, `if` statement, or "`?:`" expression, or used as an operand of "`&&`" or "`||`", is treated as "`(int)((`e`)!=0)`". This mirrors the semantics of C, and compensates for C's lack of a boolean type, while taking advantage of the ability to overload "`!=`" and "`0`".
- The controlling expression of a `switch` statement must have exactly one interpretation with an integral type, which must be unambiguous.
- Expressions that are operands of `sizeof` or "`&`" must have exactly one interpretation, which must be unambiguous.

Given the above rules, the C semantics of multiplication expressions can be specified by a set of seven predefined multiplication routines:

```
int           ?*?(int, int);                     /* ?*?_i */
unsigned int  ?*?(unsigned int, unsigned int);   /* ?*?_ui */
unsigned long ?*?(unsigned long, unsigned long); /* ?*?_ul */
long int      ?*?(long int, long int);           /* ?*?_l */
float         ?*?(float, float);                 /* ?*?_f */
double        ?*?(double, double);               /* ?*?_d */
long double   ?*?(long double, long double);     /* ?*?_ld */
```

(The subscripted names given in the comments are used to distinguish between the different routines in the examples that follow.) In the Cforall code:

```
long int a, b;
a * b;
```

"a" and "b" each have one interpretation. The multiplication expression has seven possible interpretations, one for each interpretation of "*":

```
?*?i( (int)a, (int)b);
?*?ui( (unsigned int)a, (unsigned int)b);
?*?ul( (unsigned long int)a, (unsigned long int)b);
?*?l( (long int)a, (long int)b);
?*?f( (float)a, (float)b);
?*?d( (double)a, (double)b);
?*?ld( (long double)a, (long double)b);
```

Of these, the first three involve unsafe conversions. The fourth has conversion cost 0. The fifth has cost 2 (two conversions from `long int` to `float`), and the sixth and seventh have costs 4 and 6. The expression is implicitly cast to `void`, since it occurs as a statement. This adds 1 to each interpretation's conversion cost, and produces seven interpretations with type `void`. The best of these is the fourth, since it has no unsafe conversions and has the lowest total cost. Hence, the interpretation of the statement is `(void)?*?`$_l$`( (long int)a, (long int)b)`.

Note that the C rule that applies integral promotions to "small" operands is no longer needed; it is implied by the absence of routines that have "small" parameter types. Conversely, programmers can define operators for them if they are needed.

Now, consider the bitwise complement operator, "~", which is associated with the special identifier "~?" and the following predefined routines:

```
int           ~?(int);            /* ~?i */
unsigned int  ~?(unsigned int);   /* ~?ui */
long int      ~?(long int);       /* ~?l */
unsigned long ~?(unsigned long);  /* ~?ul */
```

Given

```
void f(unsigned long int);
int  i;
f(~i);
```

the subexpression `~i` has four interpretations:

```
~?_i( (int)i ) );                  /* cost 0 */
~?_ui( (unsigned int)i ) );        /* cost 1 */
~?_l( (long int)i ) );             /* cost 1 */
~?_ul( (unsigned long int)i ) );   /* cost 2 */
```

Each interpretation must be converted to `unsigned long int` to be used as the argument of `f`. Conversions from `int` have cost 2 and from `unsigned` and `long` have cost 1, so all four possibilities for the argument of `f` contain no unsafe conversions, call no polymorphic routines, and have a total conversion cost of 2. Here the final rule (that excludes any implicit conversion applied to argument expressions) comes into play and chooses the interpretation

```
f( (unsigned long int) ~?( (int)i ) )
```

because it has the lowest conversion cost when the conversion to `unsigned long int` is ignored. The effect of this rule is to move conversions from routine arguments to routine results; C's expression semantics have the same effect.

This final rule is only necessary because all interpretations with distinct types are preserved for possible use in the next higher expression in the expression tree. An alternate overload resolution would preserve only the best of the interpretations, and would not have this rule. In the case of `f(~i)`, only the first interpretation of `~i` would be kept. However, in the case of programmer-defined overloaded routines, that scheme would not minimized conversion costs, and would even lead to unsafe conversions, as the following example shows.

```
void     f(float);
double   g(int);          /* g_i */
long int g(long int);     /* g_l */
int      v;
f(g(v));
```

In this case, g_i`(v)` has cost 0, and g_l`(v)` has cost 1. If g_l`(v)` was discarded because it has higher cost, then an unsafe conversion from `double` to `float` would be required in the call to `f`. Instead, since g_l`(v)` is preserved, it is used as the argument to `f`, and no unsafe conversions are needed.

Routines that provide operations on pointers are polymorphic. Subtraction on pointers is defined by five routines. (`ptrdiff_t` is an implementation-dependent type defined by the ANSI C standard to be some integral type that can represent the difference between any two pointers.)

```
forall(type T) const volatile T*
    ?-?(const volatile T*, ptrdiff_t);  /* ?-?_cv */
forall(type T) volatile T*
    ?-?(volatile T*, ptrdiff_t);        /* ?-?_v */
forall(type T) const T*
    ?-?(const T*, ptrdiff_t);           /* ?-?_c */
forall(type T) T*
    ?-?(T*, ptrdiff_t);                 /* ?-?_∅ */
forall(type T) ptrdiff_t
    ?-?(const volatile T*, const volatile T*);
```

Recall that in calls to polymorphic routines, type parameters are bound to unqualified object types deduced from the explicit parameters. In the case of

```
volatile int* ip;
ptrdiff_t     d;
ip - d;
```

the compiler infers that the type argument is `int`. There are no implicit conversions between pointer types and integral types, so the subtraction expression has only four possible interpretations:

```
?-?_cv( (const volatile int*)ip, (ptrdiff_t)d);
    /* returns const volatile int* */
?-?_v( (volatile int*)ip, (ptrdiff_t)d);
    /* returns volatile int* */
?-?_c( (const int*)ip, (ptrdiff_t)d);
    /* returns const int* */
?-?_∅( (int*)ip, (ptrdiff_t)d);
    /* returns int* */
```

The last two of these apply unsafe conversions to `ip`, so they are eliminated. The first two are equally polymorphic, so the choice between them depends on conversion cost. The first interpretation has conversion cost 1 (because of the implicit safe conversion from `volatile int*` to `const volatile int*`) and the second has cost 0, so the second is chosen. The result has type `volatile int*`; type matching, which is provided by the use of the type parameter `T` in

the argument and result types, preserved the `int` type, while overloading preserved the `volatile` qualifier.

C's semantics do not allow pointer subtraction on pointers to `void`, or on pointers to incomplete types. In Cforall, this restriction is a simple consequence of the use of polymorphism, since `void` and incomplete types can not be inferred as the values of type parameters. However, type qualifiers are not part of inferred type arguments, either, so four overloaded routines are needed, one for each combination of type qualifiers in the return type, to define subtraction of integers from pointers. A more serious problem is that polymorphic routines can not be used to define comparisons between pointers to incomplete types.

4.4.6 Summary of Overload Resolution

This section has discussed a set of overload resolution rules and a set of predefined routines and constants that mimic several aspects of C's expression semantics. Polymorphism allows the description of pointer operations. The application of integral promotions to operands of unary operators and usual arithmetic conversions to operands of binary operations becomes a consequence of the use of implicit conversions in routine calls. Restrictions on the set of legal operand types for operations like "`~`" are reflected directly by the set of overloadings, while restrictions that certain pointer operations apply only to pointers to object types follow from the rule that type arguments must be object types. Parametric polymorphism preserves type information in pointer operations, while overloading preserves type qualifiers. The overload resolution rules are simple compared to, for instance, those of C++, and apply uniformly to both "operators" and to programmer-defined routines.

Since C has a richer and more complicated set of conversions, operations, and data types than most other languages (thanks in part to implementation-dependent conversions), this design should be widely applicable. In fact, irregularities caused by type qualifiers, by incomplete types, and by the lack of a boolean type would not occur in other languages.

4.5 Opaque Types

Recall the principle of declaration correspondence, which states that all options or properties associated with declarations should be uniformly available. If this principle is applied to Cforall, `type` parameter declarations must be complemented by `type` identifier declarations. The semantics of Cforall's type parameters leads immediately to the following conclusions:

- `type` declarations declare identifiers that are names of object types.
- The "value" of such an identifier is constant throughout its lifetime.
- By default, instances of the type can be initialized, passed as parameters, assigned to each other, and used as operands of `sizeof` or "`&`". No other operations exist by default.
- A type declaration that contains an initializer is a *type definition*. The initial value can be any unqualified object type.
- Identifiers declared by type declarations denote types that are distinct from each other, and from their initializing type. This distinguishes them from identifiers declared by `typedef` declarations, which are synonyms for other types.

A simple example of a type declaration would be

```
type Complex = struct { double real, imaginary; };
```

which would create a new type distinct from all others (including `struct { double real, imaginary; }`). Since some way to create `Complex` values is needed, Cforall defines implicit, safe, unit-cost conversions between `Complex` and its initializing type. Since assignment of `Complex` values is allowed, Cforall uses the assignment operation of the initializing type as the default assignment operation for `Complex`.

The principle of declaration correspondence also requires that type identifiers obey the same linkage rules as ordinary identifiers. Hence it must be meaningful to have the following declarations at the file scope of a translation unit:

```
type T1 = struct { int a, b; };
static type T2 = struct { int a, b; };
extern type T3;
```

The meanings of these declarations follow from the semantics of C storage class specifiers. All three have static duration: the identifiers are bound to values that exist throughout the execution of the program. `T1` is an identifier with external linkage; it can be referred to in other translation units. `T2` is declared with internal linkage; it can be used within the translation unit that declares it, but is not visible outside of it. `T3` is also declared with external linkage, but the definition and initialization of `T3` occurs in some other translation unit. Within the scope of this declaration, the implementation of type `T3` is unknown; `T3` is an *opaque type*.

Cforall's opaque types can have simple implementations. The simplest implements them as integer variables that contain the encoding of the size and alignment of the type's implementation. The variable's storage duration and external linkage are determined by the type declaration. When the type's implementation is a compile-time constant, the Cforall compiler can initialize the variable directly. On a computer with no alignment requirements and 4-byte integers, a compiler would transform the declarations above to the equivalent of

```
int T1 = 8;
static int T2 = 8;
extern int T3;
```

In a C program, the initializer of a static definition must be a compile-time constant. For maximal simplicity, Cforall could extend this rule so that the initializer of a static type definition must also be a compile-time constant. Unfortunately, opaque types are *not* compile-time constants, because their size and alignment are not known where they are used. Hence a type definition with static duration would not be able to use opaque types in its initializer. (A pointer to an opaque type would be considered to be a compile-time constant type, and would be allowed.) This is a severe restriction: opaque types would not be able to build on other opaque types. However, it does eliminate the possibility of erroneous, mutually recursive opaque type definitions.

Cforall could avoid this restriction by removing the C rule that initializers in static-duration declarations must be compile-time constants. There are two ways to implement this. The traditional method generates a piece of code for each translation unit that initializes the static-duration storage defined in the unit, including the "type" variables that contain encoded sizes and alignments. The order in which the pieces are executed depends on the way in which translation units refer to each other; if the graph defined by the inter-unit references contains a cycle, then the program is malformed. The linker is in the best position to determine the correct order for initialization, but typical linkers do not provide this ability.

A simpler method (suggested by C. R. Zarnke) avoids the problem of initializing type encodings by implementing each opaque type definition as a routine that calls other "type" routines as necessary. The routine would have a "number of instances" parameter, so that an array of instances could be allocated with one call. The routine could simply return the encoded size and alignment. It could also allocate memory itself. The allocated memory could come from the heap; in that case memory must be scavenged when blocks exit. The routine could also allocate memory in the stack frame of the calling routine directly. The Unix `alloca()` library routine does

exactly that; in effect, the "type" routine would be a version of `alloca()`, specialized for the size to be allocated. This routine would be a sensible place to put (or call) type-specific initialization code, as well. In any case, each translation unit still needs initialization code to allocate space for instances of opaque types that have static duration.

4.5.1 Assertions and Type Declarations

Type declarations can contain assertions. In the simplest case, assertions appear in opaque type declarations, and mean that the objects and routines declared by the assertion exist.

```
extern type Complex | addable(Complex);
```

declares that a complex has a `0` object and a "+" operator; it has the same effect as

```
extern type Complex;
extern const Complex 0;
extern Complex ?+?(Complex, Complex);
```

Assertions on type definitions have a more complicated meaning. The type definition

```
type T | addable(T) = T1;
```

declares that `T` has a `0` object and "=" and "+" operators, as well as having the type `T1` for its implementation. The definitions of "`0`", "=", and "+" must occur in the scope of `T`'s definition if they are to have access to `T1`. But what if they do not? Cforall uses this as an opportunity to implement inheritance, as discussed for $F_\omega^\ni$. If an identifier is declared by an assertion declaration and is not defined by the end of `T`'s scope, Cforall attempts to build one from an object or routine with the same identifier and a similar type that is defined for `T1`. This is normally the case for assignment: `T`'s "=" calls `T1`'s "=". If there is no such identifier, and `T1` is a structure and exactly one member has an appropriate object or routine defined, that object or routine is used. In any other case, the Cforall translator will report an error.

Consider a graphics library that provides a `shape` context that describes shapes that can be drawn, a `Rect` type that represents rectangular shapes, a `Pattern` type, and a `fill` routine that fills screen areas with a pattern.

```
extern type Point;
context shape(type T) {
    void  draw(T);       /* Draw a shape.                */
    T     init(Point c,  /* Create a shape centered at c, */
               Point p); /*    with  p on its perimeter.  */
    Point center(T);     /* Return the shape's center.    */
}
extern type Rect | shape(Rect);
extern type Pattern;
extern void fill(Point pt, Pattern ptn);
```

A solid black box shape should inherit the implementation of `Rect`.

```
extern Pattern black;
type Black_box | shape(Black_box) = Rect;
void draw(Black_box b) {
    draw( (Rect)b );
    fill( center(b), black);
}
```

`Black_box` has an explicitly-defined `draw` routine. Note that the semantics of the cast operator provide access to `Rect`'s `draw`: "`(Rect)b`" converts `b` to a rectangle, and then overload resolution finds the appropriate draw. Another way to achieve same effect is to declare a routine pointer that renames `Rect`'s `draw`.

```
void draw(Black_box b) {
    void (*rect_draw)(Rect) = draw;
    rect_draw(b);
    ...
```

`init` and `center` are not defined for `Black_box`, so Cforall creates them, in effect defining

```
Black_box init(Point c, Point p) { return (Rect) init(c,p); }
Point     center(Black_box b) { return center( (Rect)b ); }
```

`Rect` could be used as the base for a new type, `Filled_rect`, which is a rectangle that is filled by a default pattern.

```
extern Pattern current_pattern;
type Filled_rect | shape(Filled_rect) = struct {
    Rect    r;
    Pattern fill_pat;
};
Filled_rect init(Point c, Point p) {
    Filled_rect f;
    f.r = init(c,p);
    f.fill_pat = current_pattern;
    return f;
}
void draw(Filled_rect fr) {
    draw(fr.r);
    fill(center(fr), fr.fill_pat);
}
```

`Filled_Rect` overrides the default versions of `init` and `draw`. ("`draw(fr.r)`" is used to call `Rect`'s `draw` inside `Filled_rect`'s `draw`.) Cforall creates a `center` routine, in effect defining

```
Point center(Filled_rect fr)  return center(fr.r);
```

4.6 Context Examples

4.6.1 C Types

This section gives examples of contexts for some groups of types that are important in the C language, in terms of the predefined operations that can be applied to those types.

Scalar, Arithmetic, and Integral Types

The pointer, integral, and floating-point types are all *scalar types*. All of these types can be logically negated and compared. The assertion "`scalar(T)`" should be read as "type `T` is scalar".

```
context scalar(type T) {
    int !?(T);
```

```
    int ?<?(T, T), ?<=?(T, T), ?==?(T, T);
    int ?>=?(T, T), ?>?(T, T), ?!=?(T, T);
};
```

The integral and floating-point types are *arithmetic types*, which support the basic arithmetic operators. The use of an assertion in the context's parameter list declares that, in order to be arithmetic, a type must also be scalar (and hence that scalar operations are available).

```
context arithmetic(type T | scalar(T) ) {
    T +?(T),     -?(T);
    T ?*?(T, T), ?/?(T, T),  ?+?(T, T),  ?-?(T, T);
};
```

The enumerated types, the `signed`, `unsigned`, and plain `char` types, and `signed` and `unsigned` `long`, `short`, and ordinary-length `int` types make up the *integral types*. They support the modulus operator and various bit-wise operations.

```
context integral(type T | arithmetic(T) ) {
    T ~?(T);
    T ?&?(T, T),  ?|?(T, T), ?^?(T, T);
    T ?%?(T, T);
    T ?<<?(T, T), ?>>?(T, T);
};
```

Modifiable Types

The only overloadable operation that can be applied to all modifiable lvalues is simple assignment. In Cforall terms, then, a modifiable type is one with an assignment operator.

```
context m_lvalue(type T) {
    T ?=?(T*, T);
};
```

Modifiable scalar lvalues are scalars and are modifiable lvalues, and assertions reflect those relationships. Modifiable scalar lvalues support the operations of scalars and modifiable lvalues, and can also be incremented and decremented.

```
context m_l_scalar(type T | scalar(T) | m_lvalue(T) ) {
    T ?++(T*), ?--(T*);
    T ++?(T*), --?(T*);
};
```

Modifiable arithmetic lvalues are both modifiable scalar lvalues and arithmetic. Note that this results in the "inheritance" of `scalar` along both paths.

```
context m_l_arithmetic(type T | arithmetic(T)
                              | m_l_scalar(T) ) {
    T ?/=?(T*, T), ?*=?(T*, T);
    T ?+=?(T*, T), ?-=?(T*, T);
};

context m_l_integral(type T | integral(T)
                            | m_l_arithmetic(T) ) {
    T ?&=?(T*, T), ?|=?(T*, T),  ?^=?(T*, T);
    T ?%=?(T*, T), ?<<=?(T*, T), ?>>=?(T*, T);
};
```

Pointer and Array Types

Array types can barely be said to exist in ANSI C, since in most cases an array name is treated as a constant pointer to the first element of the array, and the subscript expression "`a[i]`" is equivalent to the dereferencing expression "`(*(a+(i)))`". Technically, pointer arithmetic and pointer comparisons other than "`==`" and "`!=`" are only defined for pointers to array elements, but the type system does not enforce those restrictions. Consequently, there is no need for a separate "array type" context.

Pointer types are scalar types. Like other scalar types, they have "+" and "-" operators, but the types do not match the types of the operations in `arithmetic`, so these operators cannot be consolidated in `scalar`.

```
context pointer(type P | scalar(P) ) {
    P         ?+?(P, long int), ?+?(long int, P);
    P         ?-?(P, long int);
```

```
    ptrdiff_t ?-?(P, P);
};
```

Modifiable lvalue pointers support the "normal" assignment operator (provided here by "`m_l_scalar(P)`"), increment and decrement operators, and assignment to and from `void*`.

```
context m_l_pointer(type P | pointer(P) | m_l_scalar(P) ) {
    P     ?+=?(P*, long int), ?-=?(P*, long int);
    P     ?=?(P*, void*);
    void* ?=?(void**, P);
};
```

Contexts that define the dereference operator (or subscript operator) require two parameters, one for the pointer type and one for the pointed-at (or element) type. The assertion "`ptr_to(Ptr, int)`" should be read as "type `Ptr` acts like a pointer to `int`".

```
context ptr_to(type P | pointer(P), type T) {
    lvalue T *?(P);
    lvalue T ?[?](P, long int);
};
```

A pointer type can include type qualifiers in the pointed-at type. Since type arguments do not include type qualifiers, each set of qualifiers requires a different context. "`ptr_to_const(CPtr, int)`" should be read as "type `CPtr` acts like a pointer to `const int`".

```
context ptr_to_const(type P | pointer(P), type T) {
    const lvalue T *?(P);
    const lvalue T ?[?](P, long int);
};

context ptr_to_volatile(type P | pointer(P), type T) }
    volatile lvalue T *?(P);
    volatile lvalue T ?[?](P, long int);
};

context ptr_to_const_volatile(type P | pointer(P), type T) }
    const volatile lvalue T *?(P);
```

```
    const volatile lvalue T ?[?](P, long int);
};
```

Assignment to pointers is more complicated than assignment for other types, because the target's type can have more type qualifiers in the pointed-at type than the source's type does. A "`const T*`" can be assigned a "`T*`" or a "`const T*`", but not a "`volatile T*`" or a "`const volatile T*`". Contexts can model this by taking advantage of implicit conversions: there is a conversion from "`T*`" to "`const T*`", but not vice versa.

```
context m_l_ptr_to(type P | m_l_pointer(P),
                   type T | ptr_to(P,T)) {
    P  ?=?(P*, T*);
    T* ?=?(T**, P);
};

context m_l_ptr_to_const(type P | m_l_pointer(P),
                         type T | ptr_to_const(P,T)) {
    P        ?=?(P*, const T*);
    const T* ?=?(const T**, P);
};

context m_l_ptr_to_volatile(type P | m_l_pointer(P),
                            type T | ptr_to_volatile(P,T))
    P           ?=?(P*, volatile T*);
    volatile T* ?=?(volatile T**, P);
};

context m_l_ptr_to_const_volatile(
        type P | ptr_to_const_volatile(P),
        type T | m_l_ptr_to_const(P,T)
               | m_l_ptr_to_volatile(P)) {
    P                ?=?(P*, const volatile T*);
    const volatile T* ?=?(const volatile T**, P);
};
```

Consider "`m_l_ptr_to_const(Ptr, int)`", meaning "`Ptr` is a modifiable type that acts like a

pointer to `const int`". The assertion produces the declarations

```
Ptr        ?=?(Ptr*, const int*);
const int* ?=?(const int**, Ptr);
```

The first allows assignment from `const int*` to `Ptr`. It also allows assignment from `int*`, because an implicit conversion from `int*` to `const int*` exists, but it does not allow assignment from a `volatile int*`, because no `const int*` to `volatile int*` conversion exists. The second declaration allows assignment of a `Ptr` to a `const int*`. Assignment to a `const volatile int*` is defined by `m_l_ptr_to_const_volatile`, since that assignment must have a `const volatile int*` result.

Note the regular manner in which type qualifiers appear in the parameter lists of the assignment operators above. An alternative context can make use of the fact that qualification of the pointed-at type is part of a pointer type to capture that regularity.

```
context m_l_ptr_like(type MyP | m_l_pointer(MyP),
                     type CP | m_l_pointer(CP) ) {
    MyP ?=?(MyP*, CP);
    CP  ?=?(CP*, MyP);
};
```

The assertion "`| m_l_ptr_like(T, const int*)`" should be read as "`T` is a pointer type like `const int*`". This defines a relationship between two types that is quite unlike the "collection of" example used in previous chapters.

The single `m_l_ptr_like` context can replace the previous four, but compared to them it has two defects: there is no guarantee that dereferencing a `MyP` produces an lvalue of the type that `CP` points at, and the "`m_l_pointer(CP)`" assertion provides only a weak assurance that the argument passed to `CP` really is a pointer type.

4.6.2 Relationships Between Operations

Different C operators often have related meanings; for instance, "+" adds two operands, "+=" increments a variable by some amount, and the two versions of "++" increment a variable by 1 and return the variable's value before or after the increment, respectively. Languages like C++ and Ada allow programmers to define these operators for new types, but do not require that these

relationships be preserved, or even that all of the operators be implemented. Completeness and consistency is left to the good taste and discretion of the programmer. It is possible to encourage these attributes by providing (in Ada) generic operator routines, or (in C++) member functions of abstract superclasses, that are defined in terms of other, related operators.

In Cforall, polymorphic routines provide the equivalent of these generic operators, and contexts explicitly define the minimal implementation that a programmer should provide. This section shows a few examples.

Relational and Equality Operators

The different comparison operators have obvious relationships, but there is no obvious subset of the operations to use in the implementation of the others. However, it is usually convenient to implement a single comparison routine that returns a negative integer, 0, or a positive integer if its first argument is respectively less than, equal to, or greater than its second argument; the C library routine for character string comparison, `strcmp`, is one such routine. These routines can serve as the base for generic comparison routines. (Note that, compared to other programming languages, C and Cforall have an extra, non-obvious comparison operator: "`!`", logical negation, returns 1 if its operand compares equal to 0, and 0 otherwise.)

```
context comparable(type T) {
    const T 0;
    int     compare(T, T);
}

forall(type T | comparable(T)) int ?<?(T l, T r) {
    return compare(l,r) < 0;
}
/* ... similarly for <=, ==, >=, >, and !=. */

forall(type T | comparable(T)) int !?(T operand) {
    return !compare(operand, 0);
}
```

Arithmetic and Integer Operations

A complete arithmetic type would provide all of the arithmetic operators and the corresponding assignment operators. Of these, the assignment operators are more likely to be implemented directly, because it is usually more efficient to alter the contents of an existing object than to create and return a new one. Similarly, a complete integral type would provide integral operations based on integral assignment operations.

```
context arith_base(type T) {
    const T 1;
    T       ?+=?(T*,T), ?-=?(T*,T), ?*=?(T*,T), ?/=?(T*,T);
}

forall(type T | arith_base(T)) T ?+?(T l, T r) {
    return l += r;
}

forall(type T | arith_base(T)) T ?++(T* operand) {
    T temporary = *operand;
    *operand += 1;
    return temporary;
}

forall(type T | arith_base(T)) T ++?(T* operand) {
    return *operand += 1;
}
/* ... similarly for -, --, *, and /. */

context int_base(type T) {
    T ?&=?(T*, T), ?|=?(T*, T),  ?^=?(T*, T);
    T ?%=?(T*, T), ?<<=?(T*, T), ?>>=?(T*, T);
}

forall(type T | int_base(T)) T ?&?(T l, T r) {
    return l &= r;
```

```
}
/* ... similarly for |, ^, %, <<, and >>. */
```

Note that, although an arithmetic type would certainly provide comparison routines, and an integral type would provide arithmetic operations, there does not have to be any relationship among `int_base`, `arith_base` and `comparable`. Note also that these declarations provide guidance and assistance, but they do not define an absolutely minimal set of requirements. A truly minimal implementation of an arithmetic type might only provide `0`, `1`, and `?-=?`, which would be used by polymorphic `?+=?`, `?*=?`, and `?/=?` routines.

Chapter 5

Conclusions

Chapter 1 defined a number of terms used in this thesis. The most important of these is "notion", since programs contain implementations of notions that programmers find useful for solving various problems. The expressiveness of a programming language is related to the variety of notions that it can implement. The definitions were followed by a list of criteria that are useful for comparing programming languages: efficiency; support for strong, static interface checking; expressiveness and flexibility; precise interfaces, including generality and type matching; support for polymorphic data structures; separation of interfaces from implementations; support for code reuse, including generalizability and incrementality; and simplicity of the basic language.

Chapter 2 reviewed Strachey's original use of the term "polymorphism" and discussed several varieties of polymorphism. It gave the following definitions:

- Polymorphism is the ability to implement a notion so that it applies to more than one type.
- *Ad-hoc* polymorphism is present when an implementation has one or more interfaces that are defined for a set of types which need not have any common structure.
- Universal polymorphism is present when an implementation has a single interface that is applicable to a potentially infinite set of types defined by a common structure.
- A routine exhibits parametric polymorphism when the value of a parameter defines other parts of the routine's interface.
- Inclusion polymorphism is present when an identifier can be bound to values from a potentially infinite set of types with a common structure.

These definitions have the advantages of being reasonably precise and of being defined in terms of observable parts of programming languages, instead of notions or program behaviour.

The definitions were followed by a survey of a number of polymorphism mechanisms. The *ad-hoc* mechanisms surveyed were overloading, transfer functions, and set-theoretic unions. Set-theoretic unions did not appear in Cardelli and Wegner's taxonomy of polymorphism, perhaps because it is easily confused with transfer function polymorphism. All three are reasonably efficient, preserve strong static checking and separation of interfaces, and produce modest increases in generality, generalizability, and incrementality. Set-theoretic unions and overloading allow simplification of programming languages. Overloading provides type matching, while set-theoretic unions provide a weak form of polymorphic data. However, any improvements are modest because programmer-defined *ad-hoc* polymorphism applies only to an explicitly specificed, finite set of types.

Universal polymorphism mechanisms fall into two camps: inclusion polymorphism and parametric polymorphism. Inclusion polymorphism includes infinite set-theoretic unions and record subtyping. Their main drawback is that there is no way to specify precise relationships among the types of polymorphic items, since there is no name for the actual type of an argument to a polymorphic routine. Record subtyping provides greater precision than infinite unions, and provides strong static checking, separation of implementations from interfaces, generality and generalizability, simpler base languages, and polymorphic data, with reasonable efficiency. Unfortunately, it provides only weak incrementality, and contravariant routine subtyping rules limit expressiveness.

Parametric polymorphism includes universal quantification, descriptive classes, and F-bounded quantification. All three provide strong static checking, simple base languages, separation of interfaces from implementations, generality, incrementality, and type matching, with reasonable efficiency, and even polymorphic data, encoded as a polymorphic routine. F-bounded quantification conveniently packages data values with their operations, at the expense of incrementality. Descriptive classes provide greater expressiveness and incrementality than F-bounded quantification, because routines are separate from the data they manipulate and therefore can be overridden more easily. However, descriptive classes themselves are not well defined and lack the ability to define hierarchies of abstractions. For all forms of parametric polymorphism, argument inference and automatic specialization increase generality.

Chapter 3 gave precise semantics for contextual polymorphism in terms of $F_\omega^\ni$, a variant of the polymorphic typed lambda calculus F_ω. Like F_ω, it is based on the concepts of environments, types, type generators, kinds, routines, and universally quantified types. It introduced three new

concepts:

- contexts, which are abstractions of environments;
- assertions, which produce declarations by applying contexts to operands; and
- context sorts, which are analogous to routine types and type generator kinds, to control context application.

Assertions link the bodies of polymorphic abstractions to their calling environments by producing declarations that must exist in the calling environment, and provides statically checked dynamic scoping. Type generators can also use assertions to state requirements on operand types.

Contextual polymorphism provides strong static checking, separation of interfaces from implementations, precise interfaces (including type matching within an interface), generality, generalizability, and flexible. It has greater incrementality than F-bounded quantification, because routines can be overridden easily. Data polymorphism can be added gracefully through mechanisms such as existential types without losing precision in interfaces. Contexts naturally describe relationships among types, whereas F-bounds naturally describe properties of a single type. Compared to descriptive classes, contextual polymorphism provides hierarchies of contexts, and is better defined; it cleanly distinguishes types from contexts, and shows where each can be used meaningfully.

Chapter 4 discussed Cforall, an extension of the C programming language to demonstrate contexts and assertions. Cforall allows identifier overloading, with fewer restrictions on overloadable types than comparable languages. Most built-in operators are considered to be overloaded routines, and `0` and `1` are overloadable constants. The `forall` specifier defines polymorphic routines and uses assertions to constrain their argument types. Algorithms created by Cormack, Wright and Bumbulis are used to infer type arguments to polymorphic routines and to specialize polymorphic routines automatically when necessary. The existence of type parameters in polymorphic routines leads naturally to an opaque type facility.

Cforall's overload resolution rules are based on the concepts of safe conversions, conversion cost, and degrees of polymorphism. When an expression contains overloaded identifiers, the interpretation chosen minimizes the number of unsafe conversions, the degree of polymorphism of functions, and the conversion cost of the expression. The overloading rules apply uniformly to operators and ordinary routines, and were designed so that most of C's operators can be defined as a set of overloaded routines. Polymorphism is required for the definition of pointer routines. The resolution rules and definitions mimic complicated aspects of C's semantics: the integral promotions applied to instances of "small" data types in expressions, the "usual arithmetic conversions"

applied to the operands of binary operators, promotion of the results of unary operators rather than the arguments, and the operand types of the operators, including the absence of operators for pointers to incomplete types. This demonstrates the expressive power of Cforall, and is an example of the use of a small base language to define what are primitive facilities of a larger language.

5.1 Future Work

The obvious next step is to implement Cforall, to allow a "trial by fire" of contextual polymorphism. The most likely course of action would base a Cforall compiler on the GCC-2 C compiler, since that compiler provides high-quality code, portability to a number of computers, and access to a wide community of potential users. The GCC-2 compiler already extends the C language with nested routines and variable-sized arrays, which will be helpful when implementing specializations and type parameters.

A Cforall implementation should include a collection of predefined contexts and polymorphic routines. Some of the examples in section 4.6 could form the basis of this design.

The Cforall implementation will probably force changes in the language's design. Some changes will be small. For instance, the set of predefined operators might change to accommodate the predefined contexts. The definitions of the different "`?*?`" routines might be replaced by a single polymorphic routine that uses the `arithmetic` context and the "`?*=?`" operator. This change might increase the efficiency of programs that use complex, programmer-defined types: a compiler might perform inline substitution and temporary variable elimination to convert "`a*b*c`" to "`t=a; t*=b; t*=c`".

Other possible changes would deal with weaknesses in the current design.

- Cforall currently makes no provision for automatic initialization or finalization code for instances of programmer-defined types.
- Cforall does not provide type generators. If it did, polymorphic routines and type inference would combine nicely with type generators:

```
extern type Stack(type Element);
forall(type Element) extern void push(Stack(Element) *s);
```

- Cforall does not provide polymorphic data gracefully. The lambda-calculus approach that hides data inside polymorphic routines is graceless at best, and all but requires a "lambda expression" facility (which Cforall does not have) to express operations on polymorphic data. An existential type facility based on the **open** statement of section 3.2.5 may be workable.

- C has three main classes of types: object types, incomplete types, and routine types. Incomplete types and routine types do not have instantiation or assignment operations, but they can be pointed at by pointer types. Cforall's **type** identifiers only designate object types. This restricts polymorphism, since polymorphic routines can not operate on pointers to any incomplete type or to any routine types, and it increases the cost of polymorphism, since every type parameter provides instantiation and assignment even if the routine does not need them.

  Cforall could be extended with "data types" (the object and incomplete types) and "all types" (the data types and routine types). Polymorphic routines using parameters from these type classes would be more flexible and more efficient than equivalent routines using **type**. This extension would also allow even more of C's operators to be defined as Cforall routines; the equality and inequality comparisons are polymorphic over pointers to all types, and the relational operators are polymorphic over pointers to all data types.

Bibliography

[1] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. Technical Report 62, Digital Equipment Corporation Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, August 1990.

[2] American National Standards Institute, 1430 Broadway, New York, New York 10018. *American National Standard for Information Systems – Programming Language – C*, December 1989. X3.159-1989.

[3] T. P. Baker. A one-pass algorithm for overload resolution in Ada. *ACM Transactions on Programming Languages and Systems*, 4(4):601–614, October 1982.

[4] Paul G. Basset. Frame-based software engineering. *IEEE Software*, 4(4):9–16, July 1987.

[5] Hans-J. Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software—Practice and Experience*, 18(9):807–820, September 1988.

[6] Peter Bumbulis. Towards making signatures first-class. personal communication, September 1990.

[7] Conor P. Cahill. debug_malloc. comp.sources.unix, volume 22, issue 112.

[8] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.

[9] Luca Cardelli. A semantics of multiple inheritance. In G. Kahn, D. B. MacQueen, and G. D. Plotkin, editors, *Semantics of Data Types*, pages 51–67. Springer-Verlag, 1984. Lecture Notes in Computer Science v. 173.

[10] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report. Technical Report 31, Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, August 1988.

[11] Luca Cardelli and Peter Wegner. On understanding types, data abstractions, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[12] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes. In Norman Meyrowitz, editor, *OOPSLA '89 Conference Proceedings*, pages 49–70, New Orleans, Louisiana, October 1–6 1989. Association for Computing Machinery, SIGPLAN Notices 24(10).

[13] W. R. Cook. A proposal for making Eiffel type-safe. *The Computer Journal*, 32(4):305–311, 1989.

[14] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 125–135. Association for Computing Machinery, 1990.

[15] G. V. Cormack and A. K. Wright. Polymorphism in the compiled language ForceOne. In *Proceedings of the 20th Hawaii International Conference on Systems Sciences*, January 1987.

[16] G. V. Cormack and A. K. Wright. Type-dependent parameter inference. *SIGPLAN Notices*, 25(6):127–136, June 1990. Proceedings of the ACM Sigplan'90 Conference on Programming Language Design and Implementation June 20-22, 1990, White Plains, New York, U.S.A.

[17] Brad J. Cox. *Object-oriented programming; an evolutionary approach.* Addison-Wesley, 1986.

[18] O-J Dahl, B. Myhrhaug, and K. Nygaard. *Simula67 Common Base Language.* Norwegian Computing Center, Oslo Norway, October 1970.

[19] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison Wesley, first edition, 1990.

[20] Christopher W. Fraser and David R. Hanson. A retargetable compiler for ANSI C. *SIGPLAN Notices*, 26(10):29–43, October 1991.

[21] Kokichi Futatsugi, Joseph A. Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages*, pages 52–66, January 1985.

[22] Giorgio Ghelli. A static type system for message passing. In Andreas Paepcke, editor, *OOPSLA '91 Conference Proceedings*, pages 129–145, Phoenix, Arizona, October 6–11 1991. Association for Computing Machinery, SIGPLAN Notices 26(11).

[23] Carlo Ghezzi and Mehdi Jazayeri. *Programming Language Concepts.* Wiley, 1982.

[24] J.-Y. Girard. *Interpretation fonctionelle et elimination des coupures de l'arithmetique d'ordre superieur.* PhD thesis, Universite Paris, 1972.

[25] Goseph A. Goguen. Parameterized programming. *IEEE Transactions on Software Engineering*, SE-10(5):528–543, September 1984.

[26] David M. Harland. *Polymorphic Programming Languages: Design and Implementation.* Computers and their Applications. Ellis Horwood, Market Cross House, Cooper Street, Chichester, West Sussex, PO19 1EB, England, 1984.

[27] Paul N. Hilfinger. *Abstraction Mechanisms and Language Design.* ACM Distinguished Dissertations. MIT Press, 1983.

[28] C. A. R. Hoare. Hints on programming language design. Technical Report CS-73-403, Stanford University Computer Science Department, December 1973. Reprinted in [62].

[29] Jean D. Ichbiah. On the design of Ada. In R. E. A. Mason, editor, *Information Processing 83*, pages 1–10. IFIP, North-Holland, September 1983.

[30] Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report.* Springer–Verlag, third edition, 1985. Revised by Andrew B. Mickel and James F. Miner, ISO Pascal Standard.

[31] P. J. Landin. The next 700 programing languages. *Communications of the ACM*, 9:157–164, 1966.

[32] Robert G. Lanergan and Charles A. Grasso. Software engineering with reusable designs and code. In *Workshop on Reusability in Programming*, pages 224–234. ITT Programming, September 1983.

[33] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*, volume 114 of *Lecture Notes in Computer Science.* Springer-Verlag, 1981.

[34] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall International Series in Computer Science. Prentice-Hall, 1988.

[35] Robin Milner and Mads Tofte. *Commentary on Standard ML.* MIT Press, Cambridge, Massachusetts, U. S. A., 1991.

[36] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, U. S. A., 1990.

[37] James G. Mitchell, William Maybury, and Richard Sweet. Mesa language manual. Technical Report CSL–79–3, Xerox Palo Alto Research Center, April 1979.

[38] John C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 109–124. Association for Computing Machinery, January 1990.

[39] John C. Mitchell and Robert Harper. The essence of ML. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 28–46, 1988.

[40] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[41] C. Paulin-Mohring. Extracting Fw's programs from proofs in the calculus of constructions. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 89–104, Austin, TX, January 1989.

[42] Benjamin Pierce, Scott Dietzen, and Spiro Michaylov. Programming in higher-order typed lambda calculi. Technical Report CMU-CS-89-111, School of Computer Science, Carnegie Mellon University, Pittsburg, PA 15213-3890, March 1989.

[43] Benjamin C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, February 1991.

[44] John C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Colloque sur la Programmation*, pages 408–423. Springer-Verlag, 1974. Lecture Notes in Computer Science, v. 19.

[45] John C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, June 1988.

[46] David Sandberg. An alternative to subclassing. *SIGPLAN Notices*, 21(11):424–428, November 1986.

[47] David W. Sandberg. The design of the programming language X-2. Technical Report 85-60-1, Oregon State University, Department of Computer Science, Corvallis, Oregon, 97331, 1985.

[48] Mary Shaw, editor. *ALPHARD: Form and Content.* Springer-Verlag, 1981.

[49] Mary Shaw and Wm. A. Wulf. Toward relaxing assumptions in languages and their implementations. Technical report, Carnegie-Mellon University, January 1980. Reprinted in [48].

[50] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. *SIGPLAN Notices*, 21(11):38–45, November 1986.

[51] Richard M. Stallman. *GCC.* Free Software Foundation, Cambridge, MA.

[52] Standardiseringskommissionen i Sverige. *Databehandling – Programspråk – SIMULA*, 1987. Svensk Standard SS 63 61 14.

[53] Peter A. Steenkiste. The implementation of tags and run-time checking. In Peter Lee, editor, *Topics in Advanced Language Implementation*, chapter 1, pages 3–24. The MIT Press, 1991.

[54] C. Strachey. Fundamental concepts in programming languages. In *Lecture Notes for the International Summer School in Computer Programming*, Copenhagen, August 1967.

[55] Bjarne Stroustrup. What is "object-oriented programming"? In *Proceedings of the First European Conference on Object Oriented Programming*, June 1987.

[56] R. D. Tennent. Language design methods based on semantic principles. *Acta Infomatica*, 8(2):97–112, 1977. reprinted in [62].

[57] United States Department of Defense. *The Programming Language Ada: Reference Manual*, ANSI/MIL-STD-1815A-1983 edition, February 1983. Published by Springer-Verlag.

[58] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisher. Revised report on the algorithmic language ALGOL 68. *SIGPLAN Notices*, 12(5):1–70, May 1977.

[59] D. Le Verrand. *Evaluating Ada.* North Oxford Academic, 1985.

[60] Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 60–76. Association for Computing Machinery, 1989.

[61] Peter J. L. Wallis and Bernhard W. Silverman. Efficient implementation of the Ada overloading rules. *Information Processing Letters*, 10(3):120–123, April 1980.

[62] Anthony I. Wasserman, editor. *Tutorial: Programming Language Design*. Computer Society Press, 1980.

[63] B. Wegbreit. The treatment of data types in EL1. *Communications of the ACM*, 17(5):251–264, May 1974.

[64] N. Wirth. Type extensions. *ACM Transactions on Programming Languages and Systems*, 10(2):204–214, April 1988.

[65] Niklaus Wirth. On the design of programming languages. In *Information Processing 74*, pages 386–393. North Holland Publishing Company, 1974. Reprinted in [62].

[66] Andrew K. Wright. Design of the programming language ForceOne. Technical Report CS-87-10, University of Waterloo, February 1987.

[67] Benjamin Zorn and Paul Hilfinger. A memory allocation profiler for C and lisp programs. In *Summer 1988 USENIX proceedings*, 1988.

Index