Computing Surveys, Vol 17 n. 4, pp 471-522, December 1985

On Understanding Types, Data Abstraction, and Polymorphism

Luca Cardelli
AT&T Bell Laboratories, Murray Hill, NJ 07974
(current address: DEC SRC, 130 Lytton Ave, Palo Alto CA 94301)

Peter Wegner
Dept. of Computer Science, Brown University
Providence, RI 02912

Abstract

Our objective is to understand the notion of *type* in programming languages, present a model of typed, polymorphic programming languages that reflects recent research in type theory, and examine the relevance of recent research to the design of practical programming languages.

Object-oriented languages provide both a framework and a motivation for exploring the interaction among the concepts of type, data abstraction, and polymorphism, since they extend the notion of type to data abstraction and since type inheritance is an important form of polymorphism. We develop a λ-calculus-based model for type systems that allows us to explore these interactions in a simple setting, unencumbered by complexities of production programming languages.

The evolution of languages from untyped universes to monomorphic and then polymorphic type systems is reviewed. Mechanisms for polymorphism such as overloading, coercion, subtyping, and parameterization are examined. A unifying framework for polymorphic type systems is developed in terms of the typed λ-calculus augmented to include binding of types by quantification as well as binding of values by abstraction.

The typed λ-calculus is augmented by universal quantification to model generic functions with type parameters, existential quantification and packaging (information hiding) to model abstract data types, and bounded quantification to model subtypes and type inheritance. In this way we obtain a simple and precise characterization of a powerful type system that includes abstract data types, parametric polymorphism, and multiple inheritance in a single consistent framework. The mechanisms for type checking for the augmented λ-calculus are discussed.

The augmented typed λ-calculus is used as a programming language for a variety of illustrative examples. We christen this language Fun because fun instead of λ is the functional abstraction keyword and because it is pleasant to deal with.

Fun is mathematically simple and can serve as a basis for the design and implementation of real programming languages with type facilities that are more powerful and expressive than those of existing programming languages. In particular, it provides a basis for the design of strongly typed object-oriented languages.

Contents

1. From Untyped to Typed Universes

1.1. Organizing Untyped Universes

Instead of asking the question *What is a type?* we ask why types are needed in programming languages. To answer this question we look at how types arise in several domains of computer science and mathematics. The road from untyped to typed universes has been followed many times in many different fields, and largely for the same reasons. Consider, for example, the following untyped universes:

(1) Bit strings in computer memory
(2) S-expressions in pure Lisp
(3) λ-expressions in the λ-calculus
(4) Sets in set theory

The most concrete of these is the universe of bit strings in computer memory. 'Untyped' actually means that there is only one type, and here the only type is the memory word, which is a bit string of fixed size. This universe is untyped because everything ultimately has to be represented as bit strings: characters, numbers, pointers, structured data, programs, etc. When looking at a piece of raw memory there is generally no way of telling what is being represented. The meaning of a piece of memory is critically determined by an external interpretation of its contents.

Lisp's S-expressions form another untyped universe, one which is usually built on top of the previous bit-string universe. Programs and data are not distinguished, and ultimately everything is an S-expression of some kind. Again, we have only one type (S-expressions), although this is somewhat more structured (atoms and cons-cells can be distinguished) and has better properties than bit strings.

In the λ-calculus, everything is (or is meant to represent) a function. Numbers, data structures and even bit strings can be represented by appropriate functions. Yet there is only one type: the type of functions from values to values, where all the values are themselves functions of the same type.

In set theory, everything is either an element or a set of elements and/or other sets. To understand how untyped this universe is, one must remember that most of mathematics, which is full of extremely rich and complex structures, is represented in set theory by sets whose structural complexity reflects the complexity of the structures being represented. For example, integers are generally represented by sets of sets of sets whose level of nesting represents the cardinality of the integer, while functions are represented by possibly infinite sets of ordered pairs with unique first components.

As soon as we start working in an untyped universe, we begin to organize it in different ways for different purposes. Types arise informally in any domain to categorize objects according to their usage and behavior. The classification of objects in terms of the purposes for which they are used eventually results in a more or less well-defined type system. Types arise naturally, even starting from untyped universes.

In computer memory, we distinguish characters and operations, both represented as bit strings. In Lisp, some S-expressions are called lists while others form legal programs. In λ-calculus some functions are chosen to represent boolean values, others to represent integers. In set theory some sets are chosen to denote ordered pairs, and some sets of ordered pairs are then called functions.

Untyped universes of computational objects decompose naturally into subsets with uniform behavior. Sets of objects with uniform behavior may be named and are referred to as types. For example, all integers exhibit uniform behavior by having the same set of applicable operations. Functions from integers to integers behave uniformly in that they apply to objects of a given type and produce values of a given type.

After a valiant organization effort, then, we may start thinking of untyped universes as if they were typed. But this is just an illusion, because it is very easy to violate the type distinctions we have just created. In computer memory, what is the bit-wise boolean *or* of a character and a machine operation? In Lisp, what is the effect of treating an arbitrary S-expression as a program? In the λ-calculus, what is the effect of a conditional over a non-boolean value? In set theory, what is the set-union of the function successor and the function predecessor?

Such questions are the unfortunate consequence of organizing untyped universes without going all the way to typed systems; it is then meaningful to ask about the (arbitrary) representations of higher-level concepts and their interactions.

1.2. Static and Strong Typing

A type system has as its major purpose to avoid embarrassing questions about representations, and to forbid situations where these questions might come up. In mathematics as in programming, types impose constraints which help to enforce correctness. Some untyped universes, like naive set theory, were found to be logically inconsistent, and typed versions were proposed to eliminate inconsistencies. Typed versions of set theory, just like typed programming languages, impose constraints on object interaction which prevent objects (in this case sets) from inconsistent interaction with other objects.

A type may be viewed as a set of clothes (or a suit of armor) that protects an underlying untyped representation from arbitrary or unintended use. It provides a protective covering that hides the underlying representation and constrains the way objects may interact with other objects. In an untyped system untyped objects are *naked* in that the underlying representation is exposed for all to see. Violating the type system involves removing the protective set of clothing and operating directly on the naked representation.

Objects of a given type have a representation that respects the expected properties of the data type. The representation is chosen to make it easy to perform expected operations on data objects. For example,

positional notation is favored for numbers because it allows arithmetic operations to be easily defined. But there are nevertheless many possible alternatives in choosing data representations. Breaking the type system allows a data representation to be manipulated in ways that were not intended, with potentially disastrous results. For example, use of an integer as a pointer can cause arbitrary modifications to programs and data.

To prevent type violations, we generally impose a static type structure on programs. Types are associated with constants, operators, variables, and function symbols. A *type inference* system can be used to infer the types of expressions when little or no type information is given explicitly. In languages like Pascal and Ada, the type of variables and function symbols is defined by redundant declarations and the compiler can check the consistency of definition and use. In languages like ML, explicit declarations are avoided wherever possible and the system may infer the type of expressions from local context, while still establishing consistent usage.

Programming languages in which the type of every expression can be determined by static program analysis are said to be *statically typed*. Static typing is a useful property, but the requirement that all variables and expressions are bound to a type at compile time is sometimes too restrictive. It may be replaced by the weaker requirement that all expressions are guaranteed to be type-consistent although the type itself may be statically unknown; this can be generally done by introducing some run-time type checking. Languages in which all expressions are type-consistent are called strongly typed languages. If a language is strongly typed its compiler can guarantee that the programs it accepts will execute without type errors. In general, we should strive for strong typing, and adopt static typing whenever possible. Note that every statically typed language is strongly typed but the converse is not necessarily true.

Static typing allows type inconsistencies to be discovered at compile time and guarantees that executed programs are type-consistent. It facilitates early detection of type errors and allows greater execution-time efficiency. It enforces a programming discipline on the programmer that makes programs more structured and easier to read. But static typing may also lead to a loss of flexibility and expressive power by prematurely constraining the behavior of objects to that associated with a particular type. Traditional statically typed systems exclude programming techniques which, although sound, are incompatible with early binding of program objects to a specific type. For example they exclude generic procedures, e.g. sorting, that capture the structure of an algorithm uniformly applicable to a range of types.

1.3. Kinds of Polymorphism

Conventional typed languages, such as Pascal, are based on the idea that functions and procedures, and hence their operands, have a unique type. Such languages are said to be *monomorphic*, in the sense that every value and variable can be interpreted to be of one and only one type. Monomorphic programming languages may be contrasted with *polymorphic* languages in which some values and variables may have more than one type. Polymorphic functions are functions whose operands (actual parameters) can have more than one type. Polymorphic types are types whose operations are applicable to values of more than one type.

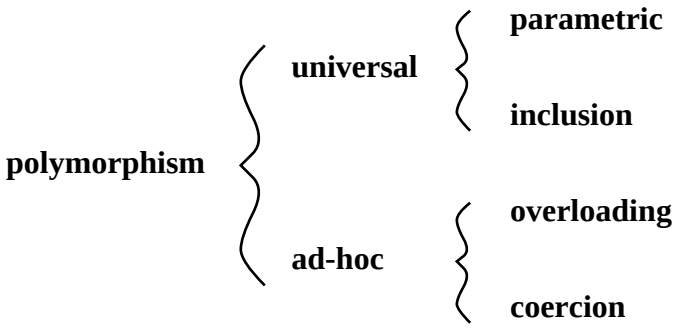

Figure 1: Varieties of polymorphism.

Strachey [Strachey 67] distinguished, informally, between two major kinds of polymorphism. *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.

Our classification of polymorphism in Figure 1 refines that of Strachey by introducing a new form of polymorphism called *inclusion polymorphism* to model subtypes and inheritance. Parametric and inclusion polymorphism are classified as the two major subcategories of *universal polymorphism*, which is contrasted with nonuniversal or ad-hoc polymorphism. Thus Figure 1 reflects Strachey's view of polymorphism but adds inclusion polymorphism to model object-oriented programming.

Parametric polymorphism is so called because the uniformity of type structure is normally achieved by type parameters, but uniformity can be achieved in different ways, and this more general concept is called *universal polymorphism*. Universally polymorphic functions will normally work on an infinite number of types (all the types having a given common structure), while an ad-hoc polymorphic function will only work on a finite set of different and potentially unrelated types. In the case of universal

polymorphism, one can assert with confidence that some values (i.e., polymorphic functions) have many types, while in ad-hoc polymorphism this is more difficult to maintain, as one may take the position that an ad-hoc polymorphic function is really a small set of monomorphic functions. In terms of implementation, a universally polymorphic function will execute the *same* code for arguments of any admissible type, while an ad-hoc polymorphic function may execute *different* code for each type of argument.

There are two major kinds of universal polymorphism, i.e., two major ways in which a value can have many types. In *parametric polymorphism*, a polymorphic function has an implicit or explicit type parameter, which determines the type of the argument for each application of that function. In *inclusion polymorphism* an object can be viewed as belonging to many different classes which need not be disjoint, i.e. there may be inclusion of classes. These two views of universal polymorphism are not unrelated, but are sufficiently distinct in theory and in practice to deserve different names.

The functions that exhibit parametric polymorphism are also called *generic functions.* For example, the length function from lists of arbitrary type to integers is called a generic length function. A generic function is one which can work for arguments of many types, generally doing the same kind of work independently of the argument type. If we consider a generic function as a single value, it has many functional types and is therefore polymorphic. Ada generic functions are a special case of this concept of *generic*.

There are also two major kinds of ad-hoc polymorphism. In *overloading* the same variable name is used to denote different functions, and the context is used to decide which function is denoted by a particular instance of the name. We may imagine that a preprocessing of the program will eliminate overloading by giving different names to the different functions; in this sense overloading is just a convenient syntactic abbreviation. A *coercion* is instead a semantic operation which is needed to convert an argument to the type expected by a function, in a situation which would otherwise result in a type error. Coercions can be provided statically, by automatically inserting them between arguments and functions at compile time, or may have to be determined dynamically by run-time tests on the arguments.

The distinction between overloading and coercion blurs in several situations. This is particularly true when considering untyped languages and interpreted languages. But even in static, compiled languages there may be confusion between the two forms of ad-hoc polymorphism, as illustrated by the following example.

```
3   + 4
3.0 + 4
3   + 4.0
3.0 + 4.0
```

Here, the ad-hoc polymorphism of + can be explained in one of the following ways:

- The operator + has four overloaded meanings, one for each of the four combinations of argument types.
- The operator + has two overloaded meanings, corresponding to integer and real addition. When one of the argument is of type integer and the other is of type real, then the integer argument is coerced to the type real.
- The operator + is defined only for real addition, and integer arguments are always coerced to corresponding reals.

In this example, we may consider the same expression as exhibiting overloading or coercion, or both (and also changing meaning), depending on an implementation decision.

Our definition of polymorphism is applicable only to languages with a very clear notion of both type and value. In particular, there must be a clear distinction between the inherent type of an object and the apparent type of its syntactic representations in languages that permit overloading and coercion. These issues are further discussed below.

If we view a type as partially specifying the behavior, or intended usage, of associated values, then monomorphic type systems constrain objects to have just one behavior, while polymorphic type systems allow values to be associated with more than one behavior. Strictly monomorphic languages are too restrictive in their expressive power because they do not allow values, or even syntactic symbols that denote values, to exhibit different behavior in different contexts of use. Languages like Pascal and Ada have ways of relaxing strict monomorphism, but polymorphism is the exception rather than the rule and we can say that they are *mostly* monomorphic. Real and apparent exceptions to the monomorphic typing rule in conventional languages include:

(1) *Overloading*: integer constants may have both type integer and real.
Operators such as + are applicable to both integer and real arguments.
(2) *Coercion*: an integer value can be used where a real is expected, and vice versa.
(3) *Subtyping*: elements of a subrange type also belong to superrange types.
(4)*Value sharing*: nil in Pascal is a constant which is shared by all the pointer types.

These four examples, which may all be found in the same language, are instances of four radically different ways of extending a monomorphic type system. Let us see how they fit in the previous description of different kinds of polymorphism.

Overloading is a purely syntactic way of using the same name for different semantic objects; the compiler can resolve the ambiguity at compile time, and then proceed as usual.

Coercion allows the user to omit semantically necessary type conversions. The required type conversions must be determined by the system, inserted in the program, and used by the compiler to generate required type conversion code. Coercions are essentially a form of abbreviation which may reduce program size and improve program readability, but may also cause subtle and sometimes dangerous system errors. The need for run-time coercions is usually detected at compile time, but languages like (impure) Lisp have plenty of coercions that are only detected and performed at run time.

Subtyping is an instance of *inclusion polymorphism*. The idea of a type being a subtype of another type is useful not only for subranges of ordered types such as integers, but also for more complex structures such as a type representing *Toyotas* which is a subtype of a more general type such as *Vehicles*. Every object of a subtype can be used in a supertype context in the sense that every Toyota is a vehicle and can be operated on by all operations that are applicable to vehicles.

Value sharing is a special case of *parametric polymorphism*. We could think of the symbol nil as being heavily overloaded, but this would be some strange kind of open-ended overloading, as nil is a valid element of an infinite collection of types which haven't even been declared yet. Moreover, all the uses of nil denote the same value, which is not the common case for overloading. We could also think that there is a different nil for every type, but all the nil's have the same representation and can be identified. The fact that an object having many types is uniformly represented for all types is characteristic of parametric polymorphism.

How do these relaxed forms of typing relate to polymorphism? As is implicit in the choice of names, universal polymorphism is considered *true* polymorphism, while ad-hoc polymorphism is some kind of *apparent* polymorphism whose polymorphic character disappears at close range. Overloading is not true polymorphism: instead of a value having many types, we allow a symbol to have many types, but the values denoted by that symbol have distinct and possibly incompatible types. Similarly, coercions do not achieve true polymorphism: an operator may appear to accept values of many types, but the values must be converted to some representation before the operator can use them; hence that operator really works on (has) only one type. Moreover, the output type is no longer dependent on the input type, as is the case in parametric polymorphism.

In contrast to overloading and coercion, subtyping is an example of true polymorphism: objects of a subtype can be uniformly manipulated as if belonging to their supertypes. In the implementation, the representations are chosen very carefully, so that no coercions are necessary when using an object of a subtype in place of an object of the supertype. In this sense the same object has many types (for example, in Simula a member of a subclass may be a longer memory segment than a member of its superclass, and its initial segment has the same structure as the member of the superclass). Similarly, operations are careful to interpret the representations uniformly so that they can work uniformly on elements of subtypes and supertypes.

Parametric polymorphism is the purest form of polymorphism: the same object or function can be used uniformly in different type contexts without changes, coercions or any kind of run-time tests or special encodings of representations. However, it should be noted that this uniformity of behavior requires that all data be represented, or somehow dealt with, uniformly (e.g., by pointers).

The four ways of relaxing monomorphic typing discussed thus far become more powerful and interesting when we consider them in connection with operators, functions and procedures. Let us look at some additional examples. The symbol + could be overloaded to denote at the same time integer sum, real sum, and string concatenation. The use of the same symbol for these three operations reflects an approximate similarity of algebraic structure but violates the requirements of monomorphism. The ambiguity can usually be resolved by the type of the immediate operands of an overloaded operator, but this may not be enough. For example, if 2 is overloaded to denote integer 2 and real 2.0, then 2+2 is still ambiguous and is resolvable only in a larger context such as assignment to a typed variable. The set of possibilities can explode if we allow user-defined overloaded operators.

Algol 68 is well known for its baroque coercion scheme. The problems to be solved here are very similar to overloading, but in addition coercions have run-time effects. A two-dimensional array with only one row can be coerced to a vector, and a vector with only one component can be coerced to a scalar. The conditions for performing a coercion may have to be detected at run time, and may actually arise from programming errors, rather than planning. The Algol 68 experience suggests that coercions should generally be explicit, and this view has been taken by many later language designs.

Inclusion polymorphism can be found in many common languages, of which Simula 67 is the earliest example. Simula's *classes* are user-defined types organized in a simple inclusion (or inheritance) hierarchy where every class has a unique immediate superclass. Simula's objects and procedures are polymorphic because an object of a subclass can appear wherever an object of one of its superclasses is required. Smalltalk [Goldberg 83], although an untyped language, also popularized this view of polymorphism. More recently, Lisp Flavors [Weinreb 81] (untyped) have extended this style of polymorphism to multiple immediate superclasses, and Amber (typed) [Cardelli 85] further extends it to higher-order functions.

The paradigmatic language for parametric polymorphism is ML [Milner 84], which was entirely built around this style of typing. In ML, it is possible to write a polymorphic identity function which works for every type of argument, and a length function which maps a list of arbitrary element type into its integer length. It is also possible to write a generic sorting package that works on any type with an ordering relation. Other languages that used or helped develop these ideas include CLU [Liskov 81], Russell [Demers 79, Hook 84], Hope [Burstall 80], Ponder [Fairbairn 82] and Poly [Matthews 85].

Finally, we should mention generic procedures of the kind found in Ada, which are parametrized templates that must be instantiated with actual parameter values before they can be used. The

polymorphism of Ada's generic procedures is similar to the parametric polymorphism of languages like ML, but is specialized to particular kinds of parameters. Parameters may be type parameters, procedure parameters, or value parameters. Generic procedures with type parameters are polymorphic in the sense that formal type parameters can take different actual types for different instantiations. However, generic type polymorphism in Ada is syntactic since generic instantiation is performed at compile time with actual type values that must be determinable (manifest) at compile time. The semantics of generic procedures is macro-expansion driven by the type of the arguments. Thus, generic procedures can be considered as abbreviations for sets of monomorphic procedures. With respect to polymorphism, they have the advantage that specialized optimal code can be generated for the different forms of inputs. On the other hand, in true polymorphic systems code is generated only once for every generic procedure.

1.3. The Evolution of Types In Programming Languages

In early programming languages, computation was identified with numerical computation and values could be viewed as having a single arithmetic type. However, as early as 1954, Fortran found it convenient to distinguish between integers and floating-point numbers, in part because differences in hardware representation made integer computation more economical and in part because the use of integers for iteration and array computation was logically different from the use of floating point numbers for numerical computation.

Fortran distinguished between integer and floating point variables by the first letter of their names. Algol 60 made this distinction explicit by introducing redundant identifier declarations for integer real and Boolean variables. Algol 60 was the first significant language to have an explicit notion of type and associated requirements for compile time type checking. Its block-structure requirements allowed not only the type but also the scope (visibility) of variables to be checked at compile time.

The Algol 60 notion of type was extended to richer classes of values in the 1960s. Of the numerous typed languages developed during this period, PL/I, Pascal, Algol 68, and Simula, are noteworthy for their contributions to the evolution of the concept of type.

PL/I attempts to combine the features of Fortran, Algol 60, Cobol, and Lisp. Its types include typed arrays, records, and pointers. But it has numerous type loopholes, such as not requiring the type of values pointed to by pointer variables to be specified, which weaken the effectiveness of compile-time type checking.

Pascal provides a cleaner extension of types to arrays records and pointers, as well as user-defined types. However, Pascal does not define type equivalence, so that the question of when two type expressions denote the same type is implementation-dependent. There are also problems with type granularity. For example, Pascal's notion of array type, which includes the array bounds as part of the type, is too restrictive in that procedures that operate uniformly on arrays of different dimensions cannot be defined. Pascal leaves loopholes in strong type specification by not requiring the full type of procedures passed as parameters to be specified, and by allowing the tag field of variant records to be independently manipulated. The ambiguities and insecurities of the Pascal type system are discussed in [Welsh 77].

Algol 68 has a more rigorous notion of type than Pascal, with a well-defined notion of type equivalence (structural equivalence). The notion of type (*mode* in Algol 68) is extended to include procedures as first-class values. Primitive modes include int, real, char, bool, string, bits, bytes, format, file, while mode constructors (type constructors) include array, struct, proc, union, and ref for respectively constructing array types, record types, procedure types, union (variant) types, and pointer types. Algol 68 has carefully defined rules for coercion, using dereferencing, deproceduring, widening, rowing, uniting, and voiding to transform values to the type required for further computation. Type checking in Algol 68 is decidable, but the type-checking algorithm is so complex that questions of type equivalence and coercion cannot always be checked by the user. This complexity was felt by some to be a flaw, resulting in a reaction against complex type systems. Thus, later languages, like Ada, had simpler notion of type equivalence with severely restricted coercion.

Simula is the first *object-oriented* language. Its notion of type includes classes whose instances may be assigned as values of class-valued variables and may persist between execution of the procedures they contain. Procedures and data declarations of a class constitute its interface and are accessible to users. Subclasses inherit declared entities in the interface of superclasses and may define additional operations and data that specialize the behavior of the subclass. Instances of a class are like data abstractions in having a declarative interface and a state that persists between invocation of operations, but lack the information-hiding mechanism of data abstractions. Subsequent object-oriented languages like Smalltalk and Loops combine the class concept derived from Simula with a stronger notion of information hiding.

Modula-2 [Wirth 83] is the first widespread language to use modularization as a major structuring principle (these ideas were first developed in Mesa). Typed interfaces specify the types and operations available in a module; types in an interface can be made *opaque* to achieve data abstraction. An interface can be specified separately from its implementation, thereby separating the specification and implementation tasks. Block-structured scoping, preserved within modules, is abandoned at a more global level in favor of flexible inter-module visibility rules achieved by import and export lists. Module interfaces are similar to class declarations (except for the above-mentioned scoping rules), but unlike class instances, module instances are not first-class values. A linking phase is necessary to interconnect module instances for execution; this phase is specified by the module interfaces but is external to the language.

ML has introduced the notion of parametric polymorphism in languages. ML types can contain type variables which are instantiated to different types in different contexts. Hence it is possible to partially specify type information and to write programs based on partially specified types which can be used on all

the instances of those types. A way of partially specifying types is just to omit type declarations: the most general (less specific) types which fit a given situation are then automatically inferred.

The above historical framework provides a basis for a deeper discussion of the relations between types, data abstraction, and polymorphism in real programming languages. We consider the untyped data abstractions (packages) of Ada, indicate the impact on methodology of requiring data abstractions to have type and inheritance, discuss the interpretation of inheritance as subtype polymorphism, and examine the relation between the subtype polymorphism of Smalltalk and the parametric polymorphism of ML.

Ada has a rich variety of modules, including subprograms to support procedure-oriented programming, packages to support data abstractions, and tasks to support concurrent programming. But it has a relatively weak notion of type, excluding procedures and packages from the domain of typed objects, and including task types relatively late in the design process as an afterthought. Its choice of name equivalence as type equivalence is weaker than the notion of structural equivalence used in Algol 68. Its severe restriction against implicit coercion weakens its ability to provide polymorphic operations applicable to operands of many types.

Packages in Ada have an interface specification of named components that may be simple variables, procedures, exceptions, and even types. They may hide a local state either by a *private* data type or in the package body. Packages are like record instances in having a user interface of named components. Ada packages differ from records in that record components must be typed values while package components may be procedures, exceptions, types, and other named entities. Since packages are not themselves types they cannot be parameters, components of structures, or values of pointer variables [Wegner 83]. Packages in Ada are second-class objects while class instances in Simula or objects in object-oriented languages are first-class objects.

The differences in behavior between packages and records in Ada is avoided in object-oriented languages by extending the notion of type to procedures and data abstractions. In the context of this discussion it is useful to define object-oriented languages as extensions of procedure-oriented languages that support typed data abstractions with inheritance. Thus we say that a language is object-oriented iff it satisfies the following requirements:

- It supports objects that are data abstractions with an interface of named operations and a hidden local state
- Objects have an associated object type
- Types may inherit attributes from supertypes

These requirements may be summarized as:

object-oriented = data abstractions + object types + type inheritance

The usefulness of this definition may be illustrated by considering the impact of each of these requirements on methodology. Data abstraction by itself provides a way of organizing data with associated operations that differs considerably from the traditional methodology of procedure- oriented programming. The realization of *data abstraction methodology* was one of the primary objectives of Ada, and this methodology is described at length in the Ada literature in publications such as [Booch 83]. However Ada satisfies only the first of our three requirements for object-oriented programming and it is interesting to examine the impact of object types and inheritance on data abstraction methodology [Hendler 86].

The requirement that all objects have a type allows objects to be first-class values so that they can be managed as data structures within the language as well as used for computation. The requirement of type inheritance allows relations among types to be specified. Inheritance may be viewed as a type composition mechanism which allows the properties of one or more types to be reused in the definition of a new type. The specification *B inherits A* may be viewed as an abbreviation mechanism which avoids redefining the attributes of type A in the definition of type B. However, inheritance is more than a shorthand, since it imposes structure among a collection of related types that can greatly reduce the conceptual complexity of a system specification. This is illustrated by the Smalltalk object hierarchy in [Goldberg 83].

The Smalltalk object hierarchy is a description of the Smalltalk programming environment in Smalltalk. It is conceptually similar to the Lisp *apply* function which describes the Lisp language interpreter in Lisp, but is a great deal more complex. It describes a collection of over 75 related system object types by an inheritance hierarchy. The object types include numerical, structured, input-output, concurrent, and display objects. The object hierarchy carefully factors out properties common to numeric objects into the supertype Number. It factors out properties common to different kinds of structured objects into the supertype Collection. It further factors out properties common to numbers, collections, and other kinds of objects into the supertype Object. In doing this the collection of over 75 object types that comprise the Smalltalk environment is described as a relatively simple structured hierarchy of object types. The shorthand provided by the object hierarchy in reusing superclasses whose attributes are shared by subclasses is clearly incidental to the conceptual parsimony achieved by imposing a coherent structure on the collection of object types.

The Smalltalk object hierarchy is also significant as an illustration of the power of polymorphism. We may characterize a polymorphic function as a function applicable to values of more than one type and *inclusion polymorphism* as a relation among types which allows operations to be applied to object of different types related by inclusion. Objects are seen as collections of such polymorphic operations

(attributes). This view emphasizes the sharing of operations by operands of many types as a primary feature of polymorphism.

The Smalltalk object hierarchy realizes polymorphism in the above sense by factoring out attributes common to a collection of subtypes into a supertype. Attributes common to numerical types are factored out into the supertype Number. Attributes common to structured types are factored out into the supertype Collection. Attributes common to all types are factored out into the supertype Object. Thus polymorphism is intimately related to the notion of inheritance, and we can say that the expressive power of object-oriented type systems is due in large measure to the polymorphism they facilitate.

In order to complete our discussion of the evolution of types in programming languages we examine the type mechanisms of ML [Milner 84]. ML is an interactive functional programming language in which type specifications omitted by the user may be reintroduced by type inference.If the user enters "3+4" the system responds "7:int", computing the value of the expression and inferring that the operands and the value are of type int. If the user enters the function declaration "fun f x = x+1" the system responds "f:int→int", defining a function value for f and inferring that it is of type "int→int". ML supports type inference not only for traditional types but also for parametric (polymorphic) types, such as the length function for lists. If "fun rec length x = if x = nil then 0 else 1+length(tail(x));" is entered, ML will infer that "length" is a function from lists of arbitrary element type to integers (length: 'a list → int). If the user then enters "length[1;2;3]", applying length to a list of integers, the system infers that length is to be specialized to the type "int list → int" and then applies the specialized function to the list of integers.

When we say that a parametric function is applicable to lists of arbitrary type we really mean that it may be specialized by (implicitly or explicitly) providing a type parameter T, and that the specialized function may then be applied to the specialized operands. There is an important distinction between the parametric function length for lists of arbitrary type and the specialized function for lists of type int. Functions like length are applicable to lists of arbitrary type because they have a uniform parametric representation that allows them to be specialized by supplying a type parameter. This distinction between a parametric function and its specialized versions is blurred in languages like ML, because type parameters omitted by the user are automatically reintroduced by the type inference mechanism.

Supertypes in object-oriented languages may be viewed as parametric types whose parameter is omitted by the user. In order to understand the similarity between parametric types and supertypes it is useful to introduce a notation where supertype parameters must be explicitly supplied in specializing a supertype to a subtype. We shall see below that Fun has explicit type parameters for both parametric types and supertypes in order to provide a uniform model for both parametric and subtype polymorphism. This results in a uniform treatment of type inference when parameters are omitted in parametric types and supertypes.

1.5. Type Expression Sublanguages

As the set of types of a programming language becomes richer, and its set of definable types becomes infinite, it becomes useful to define the set of types by a type expression sublanguage. The set of type expressions of current strongly typed programming languages is generally a simple sublanguage of the complete language that is nevertheless not altogether trivial. Type expression sublanguages generally include basic types like integer and boolean and composite types like arrays, records, and procedures constructed from basic types.

```
Type ::= BasicType | ConstructedType
BasicType ::= Int | Bool | ...
ConstructedType ::= Array(Type) | Type → Type | ...
```

The type expression sublanguage should be sufficiently rich to support types for all values with which we wish to compute, but sufficiently tractable to permit decidable and efficient type checking. One of the purposes of this paper is to examine tradeoffs between richness and tractability for type expression sublanguages of strongly typed languages.

The type expression sublanguage can generally be specified by a context-free grammar. However, we are interested not only in the syntax of the type expression sublanguage but also in its semantics. That is, we are interested in what types denote and in relations among type expressions. The most basic relation among type expressions is type equivalence. However, we are also interested in similarity relations among types that are weaker than equivalence, such as inclusion which is related to subtypes. Similarity relations among type expressions that permit a type expression to denote more than one type, or to be compatible with many types, are referred to as polymorphism.

The usefulness of a type system lies not only in the set of types that can be represented but also in the kinds of relationships among types that can be expressed. The ability to express relations among types involves some ability to perform computations on types to determine whether they satisfy the desired relationship. Such computations could in principle be as powerful as computations performable on values. However, we are concerned only with simple, easily computable relationships that express uniform behavior shared by collections of types.

The reader interested in a discussion of type expression languages and type-checking algorithms for languages like Pascal and C is referred to chapter 6 of [Aho 85], which considers type checking for overloading, coercion, and parametric polymorphism. Fun adds abstract data types to the set of basic types and adds subtype and inheritance to the forms of polymorphism that are supported.

1.6. Preview of Fun

Fun is a λ-calculus-based language that enriches the first-order typed λ-calculus with second-order features designed to model polymorphism and object-oriented languages.

Section 2 reviews the untyped and typed λ-calculus and develops first-order features of the Fun type expression sublanguage. Fun has the basic types Bool, Int, Real, String and constructed types for record, variant, function, and recursive types. This set of *first-order types* is used as a base for introducing parametric types, abstract data types, and type inheritance by means of second-order language features in subsequent sections.

Section 3 briefly reviews theoretical models of types related to features of Fun, especially models which view types as sets of values. Viewing types as sets allows us to define parametric polymorphism in terms of set intersection of associated types and inheritance polymorphism in terms of subsets of associated types. Data abstraction may also be defined in terms of set operations (in this case unions) on associated types.

Sections 4, 5, and 6 respectively augment the first-order λ-calculus with universal quantification for realizing parameterized types, existential quantification for realizing data abstraction, and bounded quantification for realizing type inheritance. The syntactic extensions of the type expression sublanguage determined by these features may be summarized as follows:

```
Type ::= ... | QuantifiedType
QuantifiedType ::=
    ∀A. Type |                              Universal Quantification
    ∃A. Type |                              Existential Quantification
    ∀A⊆Type. Type |  ∃A⊆Type. Type          Bounded Quantification
```

Universal quantification enriches the first-order λ-calculus with parameterized types that may be specialized by substituting actual type parameters for universally quantified parameters. Universally quantified types are themselves first-class types and may be actual parameters in such a substitution.

Existential quantification enriches first-order features by allowing abstract data types with hidden representation. The interaction of universal and existential quantification is illustrated in section 5.3 for the case of stacks with a universally quantified element type and an existentially quantified hidden data representation.

Fun supports information hiding not only through existential quantification but also through its let construct, which facilitates hiding of local variables of a module body. Hiding by means of let is referred to as first-order hiding because it involves hiding of local identifiers and associated values, while hiding by means of existential quantifiers is referred to as second-order hiding because it involves hiding of type representations. The relation between these two forms of hiding is illustrated in section 5.2 by contrasting hiding in package bodies with hiding in private parts of Ada packages.

Bounded quantification enriches the first-order λ-calculus by providing explicit subtype parameters. Inheritance (i.e. subtypes and supertypes) is modeled by explicit parametric specialization of supertypes to the subtype for which the operations will actually be executed. In object-oriented languages every type is potentially a supertype for subsequently defined subtypes and should therefore be modelled by a bounded quantified type. Bounded quantification provides an explanatory mechanism for object-oriented polymorphism that is cumbersome to use explicitly but useful in illuminating the relation between parametric and inherited polymorphism.

Section 7 briefly reviews type checking and type inheritance for Fun. It is supplemented by an appendix listing type inference rules.

Section 8 provides a hierarchical classification of object-oriented type systems. Fun represents the topmost (most general) type system of this classification. The relation of Fun to less general systems associated with ML, Galileo, Amber, and other languages with interesting type systems is reviewed.

It is hoped that readers will have as much fun reading about Fun as the authors have had writing about it.

2. The λ-Calculus

2.1. The Untyped λ-Calculus

The evolution from untyped to typed universes may be illustrated by the λ-calculus, initially developed as an untyped notation to capture the essence of the functional application of operators to operands. Expressions in the λ-calculus have the following syntax (we use fun instead of the traditional λ to bring out the correspondence with programming language notations):

```
e ::= x          -- a variable is a λ-expression
e ::= fun(x)e    -- functional abstraction of e
e ::= e(e)       -- operator e applied to operand e
```

The identity function and successor function may be specified in the λ-calculus as follows (with some syntactic sugar explained later). We use the keyword value to introduce a new name bound to a value or a function:

```
value id = fun(x) x              -- identity function
value succ = fun(x) x+1          -- successor function (for integers)
```

The identity function may be applied to an arbitrary λ-expression and always yields the λ-expression itself. In order to define addition on integers in the pure λ-calculus we pick a representation for integers and define the addition operation so that its effect on λ-expressions representing the integers n and m is to produce the λ-expression that represents n + m. The successor function should be applied only to λ-expressions that represent integers and suggests a notion of typing. The infix notation x+1 is an abbreviation for the functional notation + (x) (1). The symbols 1 and + above should in turn be viewed as abbreviations for a pure λ-calculus expression for the number 1 and addition.

Correctness of integer addition requires no assumptions about what happens when the λ-expression representing addition is applied to λ-expressions that do not represent integers. However, if we want our notation to have good error-checking properties, it is desirable to define the effect of addition on arguments that are not integers as an error. This is accomplished in typed programming languages by type checking that eliminates, at compilet time, the possibility of operations on objects of an incorrect type.

Type checking in the λ-calculus, just as in conventional programming languages, has the effect that large classes of λ-expressions legal in the untyped λ-calculus become illegal. The class of illegally-typed expressions depends on the type system one adopts, and, although undesirable, it may even depend on a particular type-checking algorithm.

The idea of λ-expressions operating on functions to produce other functions can be illustrated by the function twice which has the following form:

```
value twice = fun(f) fun(y) f(f(y))        -- twice function
```

The application of twice to the successor function yields a λ-expression that computes the successor of the successor.

```
twice(succ)          ⇒   fun(y) succ(succ(y))
twice (fun(x)x+1)    ⇒   fun(y) (fun(x)x+1) ((fun(x)x+1) (y))
```

The above discussion illustrates how types arise when we specialize an untyped notation such as the λ-calculus to perform particular kinds of computation such as integer arithmetic. In the next section we introduce explicit types into the λ-calculus. The resulting notation is similar to functional notation in traditional typed programming languages.

2.2. The Typed λ-Calculus

The typed λ-calculus is like the λ-calculus except that every variable must be explicitly typed when introduced as a bound variable. Thus the successor function in the typed λ-calculus has the following form:

```
value succ = fun (x: Int) x+1
```

The function twice from integers to integers has a parameter f whose type is Int → Int (the type of functions from integers to integers) and may be written as follows:

```
value twice = fun(f: Int → Int) fun (y:Int) f(f(y))
```

This notation approximates that of functional specification in typed programming languages but omits specification of the result type. We may denote the result type with a returns keyword as follows:

```
value succ = fun(x: Int) (returns Int) x + 1
```

However, the type of the result can be determined from the form of the function body x + 1. We shall omit result type specifications in the interests of brevity. Type inference mechanisms that allow this information to be recovered during compilation are discussed in a later section.

Type declarations are introduced by the keyword type. Throughout this paper, type names begin with upper-case letters while value and function names begin with lower-case letters.

```
type IntPair = Int × Int
type IntFun = Int → Int
```

Type declarations introduce names (abbreviations) for type expressions; they do not *create* new types in any sense. This is sometimes expressed by saying that we used *structural equivalence* on types instead of *name*

equivalence: two types are equivalent when they have the same structure, regardless of the names we use as abbreviations.

The fact that a value v has a type T is indicated by v:T.

```
(3,4): IntPair
succ: IntFun
```

We need not introduce variables by type declarations of the form var:T because the type of a variable may be determined from the form of the assigned value. For example the fact that intPair below has the type IntPair can be determined by the fact that (3,4) has type Int × Int, which has been declared equivalent to IntPair.

```
value intPair = (3,4)
```

However, if we want to indicate the type of a variable as part of its initialization we can do so by the notation value var:T = value.

```
value intPair: IntPair = (3,4)
value succ: Int → Int = fun(x: Int) x + 1
```

Local variables can be declared by the let-in construct, which introduces a new initialized variable (following let) in a local scope (an expression following in). The value of the construct is the value of that expression.

```
let a = 3 in a + 1          yields 4
```

If we want to specify types, we can also write:

```
let a : Int = 3 in a + 1
```

The let-in construct can be defined in terms of basic fun-expressions:

```
let a : T = M in N      ≡      (fun(a:T) N)(M)
```

2.3. Basic Types, Structured Types and Recursion

The typed λ-calculus is usually augmented with various kinds of basic and structured types. For basic types we shall use:

Unit	the trivial type, with only element ()
Bool	with an if-then-else operation
Int	with arithmetic and comparison operations
Real	with arithmetic and comparison operations
String	with string concatenation (infix) ^

Structured types can be built up from these basic types by means of type constructors. The type constructors in our language include function spaces (→), Cartesian products (×), record types (also called labeled Cartesian products) and variant types (also called labeled disjoint sums).

A pair is an element of a Cartesian product type, e.g.

```
value p = 3,true : Int × Bool
```

Operations on pairs are selectors for the first and second components:

```
fst(p)    yields  3
snd(p)    yields  true
```

A record is an unordered set of labeled values. Its type may be specified by indicating the type associated with each of its labels. A record type is denoted by a sequence of labeled types, separated by commas and enclosed in curly braces:

```
type ARecordType = {a: Int, b: Bool, c: String}
```

A record of this type may be created by initializing each of the record labels to a value of the required type. It is written as a sequence of labeled values separated by commas and enclosed in curly braces:

```
value r: ARecordType = {a = 3, b = true, c = "abcd"}
```

The labels must be unique within any given record or record type. The only operation on records is field selection, denoted by the usual *dot* notation:

```
r.b   yields   true
```

Since functions are first-class values, records may in general have function components.

```
type FunctionRecordType = {f1: Int → Int, f2: Real → Real}
value functionRecord = {f1 = succ, f2 = sin}
```

A record type can be defined in terms of existing record types by an operator & which concatenates two record types:

```
type NewFunctionRecordType = FunctionRecordType & {f3: Bool → Bool}
```

This is intended as an abbreviation, instead of writing the three fields f1, f2, and f3 explicitly. It is only valid when used on record types, and when no duplicated labels are involved.

A data structure can be made local and private to a collection of functions by let-in declarations. Records with function components are a particularly convenient way of achieving this; here is a private counter variable shared by an increment and a total function:

```
value counter =
      let count = ref(0)
      in   {increment = fun(n:Int) count := count + n,
            total = fun() count
           }

counter.increment(3)
counter.total()                yields 3
```

This example involves side-effects, as the main use of private variables is to update them privately. The primitive ref returns an updateable reference to an object, and assignments are restricted to work on such references. This is an common form of information hiding that allows updating of local state by using static scoping to restrict visibility.

A variant type is also formed from an unordered set of labeled types, which are now enclosed in brackets:

```
type AVariantType = [a: Int, b:Bool, c: String]
```

An element of this type can either be an integer labeled a, a boolean labeled b, or a string labeled c:

```
value v1 = [a = 3]
value v2 = [b = true]
value v3 = [c = "abcd"]
```

The only operation on variants is case selection. A case statement for a variant of type AVariantType, has the following form:

```
case variant of
  [a = variable of type Int] action for case a
  [b = variable of type Bool] action for case b
  [c = variable of type String] action for case c
```

where in each case a new variable is introduced and bound to the respective contents of the variant. That variable can then be used in the respective action.

Here is a function which, given an element of type AVariantType above, returns a string:

```
value f = fun (x: AVariantType)
     case x of
       [a = anInt] "it is an integer"
       [b = aBool] "it is a boolean"
       [c = aString] "it is the string: " ^ aString
     otherwise "error"
```

where the contents of the variant object x are bound to the identifiers anInt, aBool or aString depending on the case.

In the untyped λ-calculus it is possible to express recursion operators and to use them to define recursive functions. However, all computations expressible in the typed λ-calculus must terminate (roughly, the type of a function is always strictly more complex than the type of its result, hence after some number of

applications of the function we obtain a basic type; moreover, we do not have non-terminating primitives). Hence, recursive definitions are introduced as a new primitive concept. The factorial function can be expressed as:

```
rec value fact =
    fun (n:Int) if n=0 then 1 else n * fact(n-1)
```

For simplicity we assume that the only values which can be recursively defined are functions.

Finally, we introduce recursive type definitions. This allows us, for example, to define the type of integer lists out of record and variant types:

```
rec type IntList =
    [nil:Unit,
     cons: {head: Int, tail: IntList}
    ]
```

A integer list is either nil (represented as [nil = ()]) or the cons of an integer and an integer list (represented as, e.g., [cons = {head = 3, tail = nil}]).

3. Types are Sets of Values

What is an adequate notion of *type* which can account for polymorphism, abstraction and parametrization? In the previous sections we have started to describe a particular type system by giving informal typing rules for the linguistic constructs we use. These rules are enough to characterize the type system at an intuitive level, and can be easily formalized as a type inference system. The rules are sound and can stand on their own, but have been discovered and justified by studying a particular semantics of types, developed in [Hindley 69] [Milner 78] [Damas 82] [MacQueen 84a] and [Mitchell 84].

Although we do not need to discuss that semantic theory of types in detail, it may be useful to explain the basic intuitions behind it. These intuitions can in turn be useful in understanding the typing rules, particularly regarding the concept of subtypes which will be introduced later.

There is a universe V of all values, containing simple values like integers, data structures like pairs, records and variants, and functions. This is a complete partial order, built using Scott's techniques [Scott 76], but in first approximation we can think of it as just a large set of all possible computable values.

A type is a set of elements of V. Not all subsets of V are legal types: they must obey some technical properties. The subsets of V obeying such properties are called *ideals.* All the types found in programming languages are ideals in this sense, so we don't have to worry too much about subsets of V which are not ideals.

Hence, a type is an ideal, which is a set of values. Moreover, the set of all types (ideals) over V, when ordered by set inclusion, forms a lattice. The top of this lattice is the type Top (the set of all values, i.e. V itself). The bottom of the lattice is, essentially, the empty set (actually, it is the singleton set containing the least element of V).

The phrase *having a type* is then interpreted as *membership* in the appropriate set. As ideals over V may overlap, a value can have many types.

The set of types of any given programming language is generally only a small subset of the set of all ideals over V. For example any subset of the integers determines an ideal (and hence a type), and so does the set of all pairs with first element equal to 3. This generality is welcome, because it allows one to accommodate many different type systems in the same framework. One has to decide exactly which ideals are to be considered *interesting* in the context of a particular language.

A particular type system is then a collection of ideals of V, which is usually identified by giving a language of type expressions and a mapping from type expressions to ideals. The ideals in this collection are elevated to the rank of *types* for a particular language. For example, we can choose the integers, integer pairs and integer-to-integer functions as our type system. Different languages will have different type systems, but all these type systems can be built on top of the domain V (provided that V is rich enough to start with), using the same techniques.

A monomorphic type system is one in which each value belongs to at most one type (except for the least element of V which, by definition of ideal, belongs to all types). As types are sets, a value may belong to many types. A polymorphic type system is one in which large and interesting collections of values belong to many types. There is also a grey area of *mostly* monomorphic and *almost* polymorphic systems, so the definitions are left imprecise, but the important point is that the basic model of ideals over V can explain all these degrees of polymorphism.

Since types are sets, subtypes simply correspond to subsets. Moreover, the semantic assertion *T1 is a subtype of T2* corresponds to the mathematical condition $T1 \subseteq T2$ in the type lattice. This gives a very simple interpretation for subrange types and inheritance, as we shall see in later sections.

Finally, if we take our type system as consisting of the single set V, we have a type-free system in which all values have the same type. Hence we can express typed and untyped languages in the same semantic domain, and compare them.

The type lattice contains many more points than can be named in any type language. In fact it includes an uncountable number of points, since it includes every subset of the integers. The objective of a language for talking about types is to allow the programmer to name those types that correspond to interesting kinds of behavior. In order to do this the language contains type constructors, including function type constructors

(e.g., type T = T1 → T2) for constructing a function type T from domain and range types T1, T2. These constructors allow an unbound number of interesting types to be constructed from a finite set of primitive types. However, there may be useful types of the type lattice that cannot be denoted using these constructors.

In the remaining sections of this paper we introduce more powerful type constructors that allow us to talk about types corresponding to infinite unions and intersections in the type lattice. In particular, universal quantification will allow us to name types whose lattice points are infinite intersections of types, while existential quantification will allow us to name types corresponding to infinite unions. Our reason for introducing universal and existential quantification is the importance of the resulting types in increasing the expressive power of typed programming languages. It is fortunate that these concepts are also mathematically simple and that they correspond to well-known mathematical constructions.

The ideal model is not the only model of types which has been studied. With respect to other denotational models, however, it has the advantage of explaining simple and polymorphic types in an intuitive way, namely as sets of values, and of allowing a natural treatment of inheritance. Less satisfactory is its treatment of type parametrization, which is rather indirect since types cannot be values, and its treatment of type operators, which involves getting out of the model and considering functions over ideals. In view of this intuitive appeal, we have chosen the ideal model as our underlying view of types, but much of our discussion could be carried over, and sometimes even improved, if we chose to refer to other models.

The idea of types as parameters is fully developed in the second-order λ-calculus [Bruce 84]. The (only known) denotational models the second-order λ-calculus are *retract models* [Scott 76]. Here, types are not sets of objects but special functions (called retracts); these can be interpreted as identifying sets of objects, but are objects themselves. Because of the property that types are objects, retract models can more naturally explain explicit type parameters, while ideal models can more naturally explain implicit type parameters.

4. Universal Quantification

4.1. Universal Quantification and Generic Functions

The typed λ-calculus is sufficient to express monomorphic functions. However it cannot adequately model polymorphic functions. For example, it requires the previously defined function twice to be unnecessarily restricted to functions from integers to integers when we would have liked to define it polymorphically for functions a → a from an arbitrary type a to itself. The identity function can similarly be defined only for specific types such as integers: fun(x:Int)x. We cannot capture the fact that its form does not depend on any specific type. We cannot express the idea of a functional form that is the same for a variety of types, and we must explicitly bind variables and values to a specific type at a time when such binding may be premature.

The fact that a given functional form is the same for all types may be expressed by universal quantification. In particular, the identity function may be expressed as follows:

```
value id = all[a] fun(x:a) x
```

In this definition of id, a is a type variable and all[a] provides type abstraction for a so that id is the identity for all types. In order to apply this identity function to an argument of a specific type we must first supply the type as a parameter and then the argument of the given type:

```
id [Int] (3)
```

(We use the convention that type parameters are enclosed in square brackets while typed arguments are enclosed in parentheses.)

We refer to functions like id which require a type parameter before they can be applied to functions of a specific type as *generic functions*. id is the generic identity function.

Note that all is a binding operator just like fun and requires a matching actual parameter to be supplied during function application. However, all[a] serves to bind a type while fun(x:a) serves to bind a variable of a given (possibly generic) type.

Although types are applied, there is no implication that types can be manipulated as values: types and values are still distinct and type abstractions and application serve type-checking purposes only, with no run-time implications. In fact we may decide to omit the type information in square brackets:

```
value id = fun(x:a) x          where a is now a free type variable
id(3)
```

Here the type-checking algorithm has the task of recognizing that a is a free type variable and reintroducing the original all[a] and [Int] information. This is part of what a polymorphic type-checker can do, like the one used in the ML language. In fact ML goes further and allows the programmer to omit even the remaining type information:

```
value id = fun(x) x
id(3)
```

ML has a type inference mechanism that allows the system to infer the types of both monomorphic and polymorphic expressions, so that type specifications omitted by the programmer can be reintroduced by the system. This has the advantage that the programmer can use the shorthand of the untyped λ-calculus while the system can translate the untyped input into fully typed expressions. However, there are no known fully automatic type inference algorithms for the powerful type systems we are going to consider. In order for us to clarify what is happening, and not to depend on the current state of type-checking technology, we shall always write down enough type information to make the type checking task trivial.

Going back to the fully explicit language, let's extend our notation so that the type of a polymorphic function can be explicitly talked about. We denote the type of a generic function from an arbitrary type to itself by ∀a. a → a:

```
type GenericId = ∀a. a → a
id: GenericId
```

Here is an example of a function taking a parameter of a universally quantified type. The function inst takes a function of the above type and returns two instances of it, specialized for integers and booleans:

```
value inst = fun(f: ∀a. a → a) (f[Int],f[Bool])

value intid = fst(inst(id))          : Int → Int
value boolid = snd(inst(id))                                  : Bool → Bool
```

In general, function parameters of universally quantified types are most useful when they have to be used on different types in the body of a single function, e.g., a list length function passed as a parameter and used on lists of different types.

In order to show some of the freedom we have in defining polymorphic functions, we now write two versions of twice which differ in the way type parameters are passed. The first version, twice1, has a function parameter f which is of a universal type. The specification

```
fun(f: ∀a. a → a) body-of-function
```

specifies the type of function parameter f to be generic and to admit functions from any given type into the same type. Applied instances of f in the body of twice1 must have a formal type parameter f[t] and require an actual type to be supplied when applying twice1. The full specification of twice1 requires binding of the type parameter t as a universally quantified type and binding of x to t.

```
value twice1 = all[t] fun(f: ∀a. a → a) fun(x: t) f[t](f[t](x))
```

Thus twice1 has three bound variables for which actual parameters must be supplied during function application.

```
all[t]                  -- requires an actual parameter which is a type
fun(f: ∀a. a → a)       -- requires a function of the type ∀a. a → a
fun(x: t)               -- requires an argument of the type substituted for t
```

An application of twice1 to the type Int, the function id, and the argument 3 is specified as follows:

```
twice1[Int](id)(3)
```

Note that the third argument 3 has the type Int of the first argument and that the second argument id is of a universally quantified type. Note also that twice1[Int](succ) would not be legal because succ does not have the type ∀a. a → a.

The function twice2 below differs from twice1 in the type of the argument f, which is not universally quantified. Now we do not need to apply f[t] in the body of twice:

```
value twice2 = all[t] fun(f: t → t) fun(x: t) f(f(x))
twice2[Int]                yields   fun(f: Int → Int) fun(x: Int) f(f(x))
```

It is now possible to compute twice of succ:

```
twice2[Int](succ)          yields   fun(x: Int) succ(succ(x))
twice2[Int](succ)(3)       yields   5
```

Thus twice2 first receives the type parameter Int which serves to specialize the function f to be Int → Int, then receives the function succ of this type, and then receives a specific element of the type Int to which the function succ is applied twice.

An extra type application is required for twice2 of id, which has to be first specialized to Int:

```
twice2[Int](id[Int])(3)
```

Note that both λ-abstraction (function abstraction) and universal quantification (generic type abstraction) are binding operators that require formal parameters to be replaced by actual parameters. Separation between types and values is achieved by having different binding operations for types and values and different parenthesis syntax when actual parameters are supplied.

The extension of the λ-calculus to support two different kinds of binding mechanism, one for types and one for variables, is both practically useful in modeling parametric polymorphism and mathematically interesting in generalizing the λ-calculus to model two qualitatively different kinds of abstraction in the same mathematical model. In the next few sections we introduce still a third kind of abstraction and associated binding mechanism, but first we have to introduce the notion of parametric types.

In Fun, types and values are rigorously distinguished (values are objects and types are sets); hence we need two distinct binding mechanisms: fun and all. These two kinds of bindings can be unified in some type models where types are values, achieving some economy of concepts, but this unification does not fit our underlying semantics. In such models it is also possible to unify the parametric type-binding mechanism described in the next section with fun and all.

4.2. Parametric Types

If we have two type definitions with similar structure, for example:

```
type BoolPair = Bool × Bool
type IntPair = Int × Int
```

we may want to factor the common structure in a single *parametric* definition and use the parametric type in defining other types:

```
type Pair[T] = T × T
type PairOfBool = Pair[Bool]
type PairOfInt = Pair[Int]
```

A type definition simply introduces a new name for a type expression and it is equivalent to that type expression in any context. A type definition does not introduce a *new* type. Hence 3,4 is an IntPair because it has type Int × Int, which is the definition of IntPair.

A parametric type definition introduces a new *type operator*. Pair above is a type operator mapping any type T to a type T × T. Hence Pair[Int] is the type Int × Int, and it follows that 3,4 has type Pair[Int].

Type operators are not types: they operate on types. In particular, one should not confuse the following notations:

```
type A[T] = T → T
type B = ∀T. T → T
```

where A is a type operator which, when applied to a type T, gives the type of functions from T to T, and B is the type of the identity function and is never applied to types.

Type operators can be used in recursive definitions, as in the following definition of generic lists. Note that we cannot think of List[Item] below as an abbreviation which has to be *macro-expanded* to obtain the real definition (this would cause an infinite expansion). Rather, we should think of List as a new type operator which is recursively defined and maps any type to lists of that type:

```
rec type List[Item] =
      [nil: Unit,
       cons: {head: Item, tail: List[Item]}
      ]
```

A generic empty list can be defined, and then specialized, as:

```
value nil = all Item. [nil = ()]
value intNil = nil[Int]
value boolNil = nil[Bool]
```

Now, [nil = ()] has type List[Item], for any Item (as it matches the definition of List[Item]). Hence the types of the generic nil and its specializations are:

```
nil : ∀Item. List[Item]
intNil : List[Int]
boolNil : List[Bool]
```

Similarly, we can define a generic cons function, and other list operations:

```
value cons : ∀Item. (Item × List[Item]) → List[Item] =
    all Item.
        fun (h: Item, t: List[Item])
            [cons = {head = h, tail = t}]
```

Note that cons can only build homogeneous lists, because of the way its arguments and result are related by the same Item type.

We should mention that there are problems in deciding, in general, when two parametric recursive type definitions represent the same type. [Solomon 78] describes the problem and a reasonable solution which involves restricting the form of parametric type definitions.

5. Existential Quantification

Type specifications for variables of a universally quantified type have the following form, for any type expression t(a):

p: ∀a. t(a) (e.g. id: ∀a. a → a)

By analogy with universal quantification, we can try to give meaning to existentially quantified types. In general, for any type expression t(a),

p : ∃a. t(a)

has the property

For some type a, p has the type t(a)

For example:

(3,4): ∃a. a × a
(3,4): ∃a. a

where a = Int in the first case, and a = Int × Int in the second.

Thus we see that a given constant such as (3,4) can satisfy many different existential types. (Warning: for didactic purposes we assign here existential types to ordinary values, like (3,4). Although this is conceptually correct, in later sections it will be disallowed for type-checking purposes, and we shall require using particular constructs to obtain objects of existential type).

Every value has type ∃a. a because for every value there exists a type such that that value has that type. Thus the type ∃a. a denotes the set of all values, which we shall sometime call Top (the biggest type):

```
type Top = ∃a. a        -- the type of any value whatsoever
```

The set of all ordered pairs may be denoted by the following existential type.

```
∃a. ∃b. a × b           -- the type of any pair whatsoever
```

This is the type of any pair p,q because, for some type a (take a type of p) and some type b (take a type of q), p,q has type a × b.

The type of any object together with an integer-valued operation that can be applied to it may be denoted by the following existential type.

∃a. a × (a → Int)

The pair (3,succ) has this type, if we take a = Int. Similarly the pair ([1;2;3],length) has this type, if we take a = List[Int].

Because the set of types includes not only simple types but also universal types and the type Top, existentially quantified types have some properties that may at first appear counterintuitive. The type ∃a. a × a is not simply the type of pairs of equal type (e.g. 3,4), as one might expect. In fact even 3,true has this type. We know that both 3 and true have type Top; hence there is a type a = Top such that 3,true : a × a. Therefore, ∃a. a × a is the type of all pairs whatsoever, and is the same as ∃a. ∃b. a × b. Similarly, any function whatsoever has type ∃a. a → a, if we take a = Top.

However, ∃a. a × (a → Int) forces a relation between the type of an object and the type of an associated integer-valued function. For example, (3,length) does not have this type (if we consider 3 as having type Top, then we would have to show that length has type Top → Int, but we only know that length: ∀a. List[a] → a maps integer lists to integers, and we cannot assume that any arbitrary object of type Top will be mapped to integer).

Not all existential types turn out to be useful. For example, if we have an (unknown) object of type ∃a. a, we have absolutely no way of manipulating it (except passing it around) because we have no information about it. If we have an (unknown) object of type ∃a. a × a, we can assume that it is a pair and apply fst and snd to it, but then we are stuck because we have no information about a.

Existentially typed objects can be useful, however, if they are sufficiently structured. For example, x : ∃a. a × (a → Int) provides sufficient structure to allow us to compute with it. We can execute:

```
(snd(x)) (fst(x))
```

and obtain an integer.

Hence, there are *useful* existential types which hide some of the structure of the objects they represent but show enough structure to allow manipulations of the objects through operations the objects themselves provide.

These existential types can be used, for example, in forming apparently heterogeneous lists:

```
[(3,succ); ([1;2;3],length)]    : List[∃a. a × (a → Int)]
```

We can later extract an element of this list and manipulate it, although we may not know which particular element we are using and what its exact type is. Of course, we can also form totally heterogeneous lists of type List[∃a.a], but these are quite unusable.

5.1. Existential Quantification and Information Hiding

The real usefulness of existential types becomes apparent only when we realize that ∃a. a × (a → Int) is a simple example of an *abstract type* packaged with its set of operations. The variable a is the abstract type itself, which hides a representation. The representation was Int and List[Int] in the previous examples. Then a × (a → Int) is the set of operators on that abstract type: a constant of type a and an operator of type a → Int. These operators are unnamed, but we can have a named version by using record types instead of Cartesian products:

```
x: ∃a. {const: a, op: a → Int}
x.op(x.const)
```

As we do not know what the representation a really is (we only know that there is one), we cannot make assumptions about it, and users of x will be unable to take advantage of any particular implementation of a.

As we announced earlier, we have been a bit liberal in applying various operators directly to objects of existential types (like x.op above). This will be disallowed from now on, for the only purpose of making our formalism easier to type-check. Instead, we shall have explicit language constructs for creating and manipulating objects of existential types, just as we had type abstractions all[t] and type applications exp[t] for creating and using objects of universal types.

An ordinary object (3,succ) may be converted to an abstract object having type ∃a. a × (a→Int) by *packaging* it so that some of its structure is hidden. The operation pack below encapsulates the object (3,succ) so that the user knows only that an object of the type a × (a→Int) exists without knowing the actual object. It is natural to think of the resulting object as having the existential type ∃a. a × (a→Int).

```
value p = pack [a=Int in a × (a→Int)] (3,succ)   : ∃a. a × (a→Int)
```

Packaged objects such as p are called *packages*. The value (3,succ) is referred to as the *content* of the package. The type a × (a→Int) is the *interface*: it determines the structure specification of the contents and corresponds to the specification part of a data abstraction. The binding a=Int is the type *representation*: it binds the abstract data type to a particular representation Int, and corresponds to the hidden data type associated with a data abstraction.

The general form of the operation pack is as follows:

```
pack [a = typerep in interface] (contents)
```

The operation pack is the only mechanism for creating objects of an existential type. Thus, if a variable of an existential type has been declared by a declaration such as:

```
p : ∃a. a × (a → Int)
```

then p can take only values created by a pack operation.

A package must be opened before it can be used:

```
open p as x in (snd(x))(fst(x))
```

Opening a package introduces a name x for the contents of the package which can be used in the scope following in. When the structure of x is specified by labeled components, components of the opened package may be referred to by name:

```
value p = pack [a = Int in {arg:a, op:a→Int}] (3, succ)
open p as x in x.op(x.arg)
```

We may also need to refer to the (unknown) type hidden by the package. For example, suppose we wanted to apply the second component of p to a value of the abstract type supplied as an external argument. In this case the unknown type b must be explicitly referred to and the following form can be used:

```
open p as x [b] in ... fun(y:b) (snd(x))(y) ...
```

Here the type name b is associated with the hidden representation type in the scope following in. The type of the expression following in must not contain b, to prevent b from escaping its scope.

The function of open is mostly to bind names for representation types and to help the type checker in verifying type constraints. In many situations we may want to abbreviate open p as x in x.a to p.a. We are going to avoid such abbreviations to prevent confusion, but they are perfectly admissible.

Both pack and open have no run-time effect on data. Given a smart enough type checker, one could omit these constructs and revert to the notation used in the previous section.

5.2. Packages and Abstract Data Types

In order to illustrate the applicability of our notation to *real* programming languages, we indicate how records with function components may be used to model Ada packages and how existential quantification may be used to model data abstraction in Ada [D.O.D. 83]. Consider the type Point1 for creating geometric points of a globally defined type Point from pairs of real numbers and for selecting x and y coordinates of points.

```
type Point = Real × Real
type Point1  =
      {makepoint: (Real × Real) → Point,
        x_coord: Point → Real,
        y_coord: Point → Real
      }
```

Values of the type Point1 can be created by initializing each of the function names of the type Point1 to functions of the required type.

```
value point1 : Point1 =
      {makepoint = fun(x:Real,y:Real) (x,y),
       x_coord = fun(p:Point) fst(p),
       y_coord = fun(p:Point) snd(p)
      }
```

In Ada, a package point1 with makepoint, x_coord, and y_coord functions may be specified as follows:

```
package point1 is
      function makepoint (x:Real, y:Real) return Point;
      function x_coord (P:Point) return Real;
      function y_coord (P:Point) return Real;
end point1;
```

This package specification is not a type specification but part of a value specification. In order to complete the value specification in Ada, we must supply a package body of the following form:

```
package body point1 is
      function makepoint (x:Real, y:Real) return Point;
            -- implementation of makepoint
      function x_coord (P:Point) return Real;
            -- implementation of x_coord
      function y_coord (P:Point) return Real;
            -- implementation of y_coord
end point1;
```

The package body supplies function bodies for function types of the package specification. In contrast to our notation, which allows different function bodies to be associated with different values of the type,

Ada does not allow packages to have types, and directly defines the function body for each function type in the package body.

Packages allow the definition of groups of related functions that share a local hidden data structure. For example a package localpoint with a local data structure point has the following form:

```
package body localpoint is
    point: Point;   -- shared global variable of makepoint, x_coord, y_coord
    procedure makepoint(x,y: Real); ...
    function x_coord return Real; ...
    function y_coord return Real; ...
end localpoint;
```

Hidden local variables can be realized in our notation by the let construct:

```
value localpoint =
    let p: Point = ref((0,0))
    in  {makepoint = fun(x: Real, y: Real) p := (x, y),
         x_coord = fun() fst(p),
         y_coord = fun() snd(p)
        }
```

Although Ada does not have the concept of a package type it does have the notion of a package template, which has some, but not all, the properties of a type. Package templates are introduced by the keyword generic.

```
generic
    package Point1 is
        function makepoint (x:Real, y:Real) return Point;
        function x_coord (P:Point) return Real;
        function y_coord (P:Point) return Real;
    end Point1;
```

Values point1 and point2 of the generic package template Point1 can be introduced as follows:

```
package point1 is new Point1;
package point2 is new Point1;
```

All package values associated with a given generic package template have the same package body. The specification of an Ada package is statically associated with its body prior to execution, while the typed values of record types are dynamically associated with function bodies when the value-creation command is executed.

Components of package values created from a generic package can be accessed using the record notation.

```
type p is Point;
p = point1.makepoint(3,4);
```

Thus packages are like record values in allowing their components to be accessed by the same notation as is used for selection of record components. But packages are not first-class values in Ada. They cannot be passed as parameters of procedures, cannot be components of arrays or record data structures, and cannot be assigned as values of package variables. Moreover, generic package templates are not types, although they are like types in allowing instances to be created. In effect, Ada has two similar but subtly different language mechanisms for handling record-like structures, one for handling data records with associated record types, and one for handling packages with associated generic templates. By contrasting the two mechanisms of Ada for record types and generic packages with the single mechanism of our notation we gain appreciation of and insight into the advantages of uniformly extending types to records with function components.

Ada packages which simply encapsulate a set of operations on a publicly defined datatype, do not need fancy type operators. They can be modelled in our notation by the simple typed λ-calculus without existential quantification. It is only when we hide the type representation using private data types that existential quantification is needed.

The let construct was used in the previous example to realize information hiding. We call this *first order* information hiding because it is achieved by restricting scoping at the value level. This is contrasted to *second order* information hiding that is realized by existential quantifiers, which restrict scoping at the type level.

An Ada point package point2 with a private type Point may be defined as follows:

```
package point2
    type Point is private;
```

```
        function makepoint (x:Real, y:Real) return Point;
        function x_coord (P:Point) return Real;
        function y_coord (P:Point) return Real;
        private
            -- hidden local definition of the type Point
    end point2;
```

The private type Point may be modelled by existential quantification:

```
    type Point2 =
        ∃Point.
            {makepoint: (Real × Real) → Point,
             x_coord: Point → Real,
             y_coord: Point → Real
            }
```

It is sometimes convenient to view the type specifications of an existentially quantified type as a parametric function of the hidden type parameter. In the present example we may define Point2WRT[Point] as follows:

```
    type Point2WRT[Point] =
        {makepoint: (Real × Real) → Point,
         x_coord: Point → Real,
         y_coord: Point → Real
        }
```

The notation WRT in Point2WRT[Point], to be read as *with respect to*, underlines the fact that this type specification is relative to a type parameter.

A value point2 of the existential type Point2 may be created by the pack operation.

```
    value point2 : Point2 = pack [Point = (Real × Real) in Point2WRT[Point]]
        point1
```

The pack operation hides the representation Real × Real of Point, has the existentially parametrized type Point2WRT[Point] as its specification part, and provides as its hidden body the previously defined value point1 that implements operations for the given data representation.

Note that Point2WRT[Point] represents a parameterized type expression which, when supplied with an actual type parameter such as Real, determines a type (in this case a record type with three components). The relation between this kind of parameterization and the other kinds of parameterization introduced so far is illustrated by the following table:

1. Function abstraction: fun(x: type) value-expr(x). The parameter x is a value and the result of substituting an actual parameter for the formal parameter determines a value.

2. Quantification: all(a) value-expr(a). The parameter a is a type and the result of substituting an actual type for the formal parameter determines a value.

3. Type Abstraction: TypeWRT[T] = type-expr(T). The parameter T is a type and the result of substituting an actual type for the formal parameter is also a type.

Actual type parameters are restricted to be types, while actual value parameters may be arbitrarily complex values. However, when the class of namable types is enriched to include universally and existentially quantified types, this also enriches the arguments that may be substituted for formal type parameters.

Existential quantification can be used to model the private types of Ada. However, it is much more general than the data abstraction facility of Ada, as shown in the following examples.

5.3. Combining Universal and Existential Quantification

In this section we give an example that demonstrates the interaction between universal and existential quantification. Universal quantification yields generic types while existential quantification yields abstract data types. When these two notions are combined we obtain parametric data abstractions.

Stacks are an ideal example to illustrate the interaction between generic types and data abstraction. The simplest form of a stack has both a specific element type such as *integer* and a specific data structure implementation such as a *list* or an *array*. Generic stacks parameterize the element type, while abstraction from the data representation may be accomplished by creating a package that has an existential data type. A stack with parameterized element type and a hidden data representation is realized by combining universal quantification to realize the parameterization with existential quantification to realize the data abstraction.

The following operations on lists and arrays will be used:

```
nil:      ∀a. List[a]
cons:     ∀a. (a × List[a]) → List[a]
hd:       ∀a. List[a] → a
tl:       ∀a. List[a] → List[a]
null:     ∀a. List[a] → Bool

array:    ∀a. Int → Array[a]
index:    ∀a. (Array[a] × Int) → a
update:   ∀a. (Array[a] × Int × a) → Unit
```

We start with a concrete type IntListStack with integer elements and a list data representation. This concrete type can be implemented as a tuple of operations with no quantification.

```
type IntListStack =
      {emptyStack: List[Int],
       push: (Int × List[Int]) → List[Int],
       pop: List[Int] → List[Int],
       top: List[Int] → Int
      }
```

An instance of this stack type with components initialized to specific function values may be defined as follows:

```
value intListStack : IntListStack =
      {emptyStack = nil[Int],
       push = fun(a:Int,s:List[Int]) cons[Int](a,s),
       pop = fun(s:List[Int]) tl[Int](s),
       top = fun(s:List[Int]) hd[Int](s)
      }
```

We could also have a stack of integers implemented via pairs consisting of an array and a top-of-stack index into the array; this concrete stack may again be implemented as a tuple without any quantification.

```
type IntArrayStack =
      {emptyStack: (Array[Int] × Int),
       push: (Int × (Array[Int] × Int)) → (Array[Int] × Int),
       pop: (Array[Int] × Int) → (Array[Int] × Int),
       top: (Array[Int] × Int) → Int
      }
```

An instance of IntArrayStack is an instance of the above tuple type with operation fields initialized to operations on the array stack representation.

```
value intArrayStack : IntArrayStack =
      {emptyStack = (Array[Int](100),-1),
       push = fun(a:Int,s:(Array[Int] × Int))
                 update[Int](fst(s),snd(s)+1,a); (fst(s),snd(s)+1),
       pop = fun(s:(Array[Int] × Int)) (fst(s),snd(s)-1),
       top = fun(s:(Array[Int] × Int)) index[Int](fst(s),snd(s))
      }
```

The concrete stacks above may be generalized both by making the element type generic and by hiding the stack data representation. The next example illustrates how a generic element type may be realized by universal quantification. We first define the type GenericListStack as a universally quantified type:

```
type GenericListStack =
      ∀Item.
          {emptyStack: List[Item],
           push: (Item × List[Item]) → List[Item],
           pop: List[Item] → List[Item],
           top: List[Item] → Item
          }
```

An instance of this universal type may be created by universal quantification of a record instance whose fields are initialized to operations parameterized by the generic universally quantified parameter.

```
value genericListStack : GenericListStack =
    all[Item]
        {emptyStack = nil[Item],
         push = fun(a:Item,s:List[Item]) cons[Item](a,s),
         pop = fun(s:List[Item]) tl[Item](s),
         top = fun(s:List[Item]) hd[Item](s)
        }
```

The genericListStack has, as its name implies, a concrete list implementation of the stack data structure. An alternative type GenericArrayStack with a concrete array implementation of the stack data structure may be similarly defined:

```
type GenericArrayStack =  ...

value genericArrayStack : GenericArrayStack = ...
```

Since the data representation of stacks is irrelevant to the user, we would like to hide it so that the stack interface is independent of the hidden stack data representation. We would like to have a single type GenericStack which can be implemented as a generic list stack or a generic array stack. Users of GenericStack should not have to know which implementation of GenericStack they are using.

This is where we need existential types. For any item type there should exist an implementation of stack which provides us with stack operations. This results in a type GenericStack defined in terms of a universally quantified parameter Item and an existentially quantified parameter Stack as follows:

```
type GenericStack =
    ∀Item. ∃Stack. GenericStackWRT[Item][Stack]
```

The two-parameter type GenericStackWRT[Item][Stack] may in turn be defined as a tuple of doubly parameterized operations:

```
type GenericStackWRT[Item][Stack] =
    {emptystack: Stack,
     push: (Item,Stack) → Stack,
     pop: Stack → Stack,
     top: Stack → Item
    }
```

Note that there is nothing in this definition to distinguish the rôle of the two parameters Item and Stack. However, in the definition of GenericStack the parameter Item is universally quantified, indicating that it represents a generic type, while the parameter Stack is existentially quantified, indicating that it represents a hidden abstract data type.

We can now abstract our genericListStack and genericArrayStack packages into packages of type GenericStack:

```
value listStackPackage : GenericStack =
    all[Item]
        pack[Stack = List[Item] in GenericStackWRT[Item][Stack]]
            genericListStack[Item]

value arrayStackPackage : GenericStack =
    all[Item]
        pack[Stack = (Array[Item] × Item) in GenericStackWRT[Item][Stack]]
            genericArrayStack[Item]
```

Both listStackPackage and arrayStackPackage have the same type and differ merely in the form of the hidden data representation.

Moreover, functions like the following useStack can work without any knowledge of the implementation:

```
value useStack =
    fun(stackPackage: GenericStack)
        open stackPackage[Int] as p [stackRep]
        in p.top(p.push(3,p.emptystack));
```

and can be given any implementation of GenericStack as a parameter:

```
useStack(listStackPackage)
useStack(arrayStackPackage)
```

In the definition of GenericStack, the type Stack is largely unrelated to Item, while it is our intention that, whatever the implementation of Stack, stacks should be collections of items (actually, there is a weak dependency of Stack upon Item given by the order of the quantifiers). Because of this, it is possible to build objects of type GenericStack where stacks have nothing to do with items, and do not obey properties like pop(push(a,s)) = a. This limitation is corrected in more powerful type systems like [MacQueen 86] and [Burstall 84], where it is possible to abstract on type operators (e.g. List) instead of just types (e.g. List[Int]), and one can directly express that representations of Stack must be based on Item (but even in those more expressive type systems it is possible to *fake* stack packages which do not obey stack properties).

5.4. Quantification and Modules

We are now ready for a major example: geometric points. We introduce an abstract type with operations mkpoint (make a new point from two real numbers), x-coord and y-coord (extract the x and y coordinates of a point):

```
type Point =
    ∃PointRep.
        {mkpoint: (Real × Real) → PointRep,
         x-coord: PointRep → Real,
         y-coord: PointRep → Real
        }
```

Our purpose is to define values of this type that hide both the representation PointRep and the implementation of the operations mkpoint, x-coord, and y-coord with respect to this representation. In order to accomplish this we define the type of these operations as a parametric type with the point representation PointRep as a parameter. The type name PointWRT emphasizes that the operations are defined with respect to a particular representation and that in contrast the abstract datatype Point is representation-independent.

```
type PointWRT[PointRep] =
    {mkpoint: (Real × Real) → PointRep,
     x-coord: PointRep → Real,
     y-coord: PointRep → Real
    }
```

The existential type Point may be defined in terms of PointWRT by existential abstraction with respect to PointRep:

```
type Point = ∃PointRep. PointWRT[PointRep]
```

The relationship between representation-dependent point operations and the associated abstract datatype becomes even clearer when we illustrate the abstraction process for some specific point representations. Let's define a Cartesian point package whose point representation is by pairs of reals and whose operations mkpoint, x-coord, y-coord are as follows:

```
value cartesianPointOps =
    {mkpoint = fun (x:Real, y:Real) (x,y),
     x-coord = fun (p: Real × Real) fst(p),
     y-coord = fun (p: Real × Real) snd(p)
    }
```

A package with point representation Real × Real and with the above implementations of point operations as its content can be specified as follows:

```
value cartesianPointPackage =
    pack [PointRep = Real × Real in PointWRT[PointRep]]
        cartesianPointOps
```

Similarly we can make a polar point package whose point representation Real × Real is the same as that for the Cartesian point package but whose content is a different (polar-coordinate) implementation of the operations:

```
value polarPointPackage =
    pack[PointRep = Real × Real  in  PointWRT[PointRep]]
        {mkpoint = fun (x:Real, y:Real) ... ,
         x-coord = fun (p: Real × Real) ... ,
         y-coord = fun (p: Real × Real) ...
```

```
        }
```

These examples illustrate how a package realizes data abstraction by hiding both the data representation and the implementation of its operations. The Cartesian and polar packages have the same existential type Point, use the same parametric type PointWRT[PointRep] to specify the structure of point operations, and have the same type Real × Real for data representation. They differ only in the content of the package that determines the function implementations. In general, a given existential type forces all packages of that type to have the same structure for operations. But both the type of the internal data representation and the value (implementation) of the operations may differ for different realizations of an abstract data type.

An abstract data type packaged with its operators, like Point, is also a simple example of a *module*. In general modules can import other (known) modules, or can be parameterized with respect to other (as yet unknown) modules.

Parametric modules can be treated as functions over existential types. Here is a way of extending the Point package with another operation (add). Instead of doing this extension for a particular Point package, we write a procedure to do the extension for any Point package over an unknown representation of point. Recall that & is the record type concatenation operator:

```
type ExtendedPointWRT[PointRep] =
      PointWRT[PointRep] & {add: (PointRep × PointRep) → PointRep}

type ExtendedPoint = ∃PointRep. ExtendedPointWRT[PointRep]

value extendPointPackage =
      fun (pointPackage: Point)
          open pointPackage as p [PointRep] in
                pack[PointRep' = PointRep in ExtendedPointWRT[PointRep']]
                    p &
                        {add = fun (a:PointRep, b:PointRep)
                         p.mkpoint(p.x-coord(a)+p.x-coord(b),
                         p.y-coord(a)+p.x-coord(b))
                        }

value extendedCartesianPointPackage =
      extendPointPackage(cartesianPointPackage)

value extendedPolarPointPackage =
      extendPointPackage(polarPointPackage)
```

We now go back to the Point module and show how other modules can be built on top of it. In particular, we build modules Circle and Rectangle on top of Point and then define a module Picture which uses both Circle and Rectangle. As different instances of Point may be based on different data representations, we have to make sure that circles and rectangles are based on the same representation of Point, if we want to make them interact.

A circle package provides operations to create a circle out of a point (the center) and a real (the radius), and operations to extract the center and the radius of a circle. An operation diff of circle difference (distance between the centers of two circles) is also defined. The two parameters to diff are circles based on the same implementation of Point. A circle package also provides a point package, to allow one to access point operations working on the same representation of point used in the circle package.

```
type CircleWRT2[CircleRep,PointRep] =
      {pointPackage: PointWRT[PointRep],
       mkcircle: (PointRep × Real) → CircleRep,
       center: CircleRep → PointRep,
       radius: CircleRep → Real,
       diff: (CircleRep × CircleRep) → Real
      }

type CircleWRT1[PointRep] =
      ∃CircleRep. CircleWRT2[CircleRep,PointRep]

type Circle = ∃PointRep. CircleWRT1[PointRep]

type CircleModule =
      ∀PointRep. PointWRT[PointRep] → CircleWRT1[PointRep]

value circleModule : CircleModule =
      all[PointRep]
```

```
        fun (p: PointWRT[PointRep])
            pack[CircleRep = PointRep × Real in CircleWRT2[CircleRep,PointRep]]
                {pointPackage = p,
                 mkcircle = fun (m:PointRep,r:Real) (m,r),
                 center = fun (c: PointRep × Real) fst(c),
                 radius = fun (c: PointRep × Real) snd(c),
                 diff = fun (c1: PointRep × Real, c2: PointRep × Real)
                            let p1 = fst(c1)
                            and p2 = fst(c2)
                            in sqrt((p.x-coord(p1) - p.x-coord(p2))**2+
                                 (p.y-coord(p1) - p.y-coord(p2))**2)
                }
```

We can now build some particular circle packages by applying circleModule to various point packages. We could also define different versions of circleModule based on different representations of circle, and all of those could be applied to all the different point packages to obtain circle packages. Here we apply circleModule to cartesianPointPackage and to polarPointPackage to obtain cartesian and polar circle packages.

```
value cartesianCirclePackage =
    open cartesianPointPackage as p [Rep] in
        pack[PointRep = Rep in CircleWRT1[PointRep]]
            circleModule[Rep](p)

value polarCirclePackage =
    open polarPointPackage as p [Rep] in
        pack[PointRep = Rep in CircleWRT1[PointRep]]
            circleModule[Rep](p)
```

To use a circle package we have to open it. We actually have to open it twice (note that the type Circle has a double existential quantification) to bind PointRep and CircleRep to the point and circle representations used in that package. Here we use an abbreviated form of open which is equivalent to two consecutive opens:

```
open cartesianCirclePackage as c [PointRep] [CircleRep]
in ... c.mkcircle(c.pointPackage.mkpoint(3,4),5) ...
```

A rectangle is determined by two points: the upper left and the bottom right corner. The definition of rectangle module is very similar to that of the circle module. In addition, we have to make sure that the two points determining a rectangle are based on the same representation of Point.

```
type RectWRT2[RectRep,PointRep] =
    {pointPackage: PointWRT[PointRep],
     mkrect: (PointRep × PointRep) → RectRep,
     toplft: RectRep → PointRep,
     botrht: RectRep → PointRep
    }

type RectWRT1[PointRep] =
    ∃RectRep. RectWRT2[RectRep,PointRep]

type Rect = ∃PointRep. RectWRT1[PointRep]

type RectModule =
    ∀PointRep. PointWRT[PointRep] → RectWRT1[PointRep]

value rectModule =
    all[PointRep]
        fun (p: PointWRT[PointRep])
            pack[PointRep = PointRep in RectWRT1[PointRep]]
                {pointPackage = p,
                 mkrect = fun (tl: PointRep, br: PointRep) (tl,br),
                 toplft = fun (r: PointRep × PointRep) fst(r),
                 botrht = fun (r: PointRep × PointRep) snd(r)
                }
```

We now put it all together in a module of figures, which uses circles and rectangles (based on the same implementation of point) and defines an operation boundingRect which returns the smallest rectangle containing a given circle.

```
type FiguresWRT3[RectRep,CircleRep,PointRep] =
      {circlePackage: CircleWRT[CircleRep,PointRep]
       rectPackage: RectWRT[RectRep,PointRep]
       boundingRect: CircleRep → RectRep
      }

type FiguresWRT1[PointRep] =
      ∃RectRep. ∃CircleRep. FigureWRT3[RectRep,CircleRep,PointRep]

type Figures = ∃PointRep. FigureWRT1[PointRep]

type Figures =
      ∀PointRep. PointWRT[PointRep] → FiguresWRT1[PointRep]

value figuresModule =
      all[PointRep]
          fun (p: PointWRT[PointRep])
                pack[PointRep = PointRep in FiguresWRT1[PointRep]]
                    open circleModule[PointRep](p) as c [CircleRep]
                    in   open rectModule[PointRep](p) as r [RectRep]
                         in    {circlePackage = c,
                                rectPackage = r,
                                boundingRect =
                                    fun(c: CircleRep) ..r.mkrect(..c.center(c)..)..
                               }
```

5.5. Modules are First-Class Values

In the previous section we have shown that packages and modules are first-class citizens: they are legal values which can be passed and returned from functions and stored in data structures. For example, it is possible to write programs which, depending on conditions, produce one or another package of the same existential type implementing an interface, and return it to be used in the construction of larger packages.

The process of linking modules can also be expressed: we have done this in the previous example, e.g., when we produced cartesianCirclePackage by linking cartesianPointPackage and circleModule. Hence, the process of building systems out of modules can be expressed in the same language used to program modules, and the full power of the language can be applied during the linking phase.

Although we have shown that we can express parametric modules and linking mechanisms, we do not claim that this is the most convenient notation to work with. Our purpose is to show that all these concepts can be captured in a relatively simple framework. There is more to be done, however, to prevent the notation from getting out of hand. The major problem here is that one must be aware of the dependency graph of modules when creating new module instances, and the linking must be done *by hand* for every new instance. These problems are particularly addressed in the Standard ML module mechanism [MacQueen 84b].

6. Bounded Quantification

6.1. Type Inclusion, Subranges, and Inheritance

We say that a type A *is included in*, or *is a subtype of* another type B when all the values of type A are also values of type B, i.e. exactly when A, considered as a set of values, is a subset of B. This general notion of inclusion specializes to different inclusion rules for different type constructors. In this section we discuss inclusions of subranges, records, variants and function types. Inclusions of universally and existentially quantified types are discussed in later sections.

As an introduction to inclusions on record types, we first present a simple theory of inclusions on integer subrange types. Let n..m denote the subtype of the type Int associated with the subrange n to m, extremes included, where n and m are known integers. The following type inclusion relations hold for integer subrange types:

$n..m \le n'..m'$ iff $n' \le n$ and $m \le m'$

where the ≤ on the left is type inclusion and those on the right are *less or equal to*.

Subrange types may occur as type specifications in λ-expressions:

```
value f = fun (x: 2..5) x + 1
f : 2..5 → 3..6
f(3)
```

The constant 3 has the type 3..3 and also has the type of any supertype, including the type 2..5 of x above. It is therefore a legal argument of f. Similarly the following should be legal:

```
value g = fun (y: 3..4) f(y)
```

as the type of y is a subtype of the domain of f. An actual parameter of an application can have any subtype of the corresponding formal parameter.

Consider a function of type $3..7 \to 7..9$. This can also be considered a function of type $4..6 \to 6..10$, as it maps integers between 3 and 7 (and hence between 4 and 6) to integers between 7 and 9 (and hence between 6 and 10). Note that the domain shrinks while the codomain expands. In general we can formulate the inclusion rules for functions as follows:

$$s \to t \;\leq\; s' \to t' \quad \text{iff} \quad s' \leq s \;\text{ and }\; t \leq t'$$

note the (rather accidental) similarity of this rule and the rule for subranges, and how the inclusion on the domain is swapped.

The interesting point of these inclusion rules is that they also work for higher functional types. For example:

```
value h = fun (f: 3..4 → 2..7) f(3)
```

can be applied to f above:

```
h(f)
```

because of the inclusion rules for subranges, arrows and application.

The same line of reasoning applies to record types. Suppose we have types:

```
type Car = {age:Int, speed:Int, fuel:String}
type Vehicle = {age:Int, speed:Int}
```

We would like to claim that all cars are vehicles, i.e. that Car is a subtype of Vehicle. To achieve this we need the following inclusion rule for record types:

$$\{a_1{:}t_1, \ldots, a_n{:}t_n, \ldots, a_m{:}t_m\} \leq \{a_1{:}u_1, \ldots, a_n{:}u_n\}$$
$$\text{iff} \quad t_i \leq u_i \quad \text{for} \quad i \in 1..n.$$

i.e., a record type A is a subtype of another record type B if A has all the attributes (fields) of B, and possibly more, and the types of the common attributes are respectively in the subtype relation.

The meaning of the type Vehicle is the set of all records that have at least an integer field age and an integer field speed, and possibly more. Hence any car is in this set, and the set of all cars is a subset of the set of all vehicles. Again, subtypes are subsets.

Subtyping on record types corresponds to the concept of inheritance (subclasses) in languages, especially if records are allowed to have functional components. A class instance is a record with functions and local variables, and a subclass instance is a record with at least those functions and variables, and possibly more.

In fact we can also express multiple inheritance. If we add the type definitions:

```
type Object = {age:Int}
type Machine = {age:Int, fuel:String}
```

then we have that car is a subtype (inherits properties from) both vehicle and machine, and those are both subtypes of object. Inheritance on records also extends to higher functional types, as in the case of subranges, and the inclusion rule for function spaces is also maintained.

In the case of variant types, we have the following inclusion rule:

$$[a_1{:}t_1, .. , a_n{:}t_n] \le [a_1{:}u_1, .. , a_n{:}u_n, .. , a_m{:}u_m]$$
$$\text{iff} \quad t_i \le u_i \quad \text{for} \quad i \in 1..n.$$

For example, every bright color is a color:

```
type brightColor = [red:Unit, green:Unit, blue:Unit]
type color = [red:Unit, green:Unit, blue:Unit, gray:Unit, brown:Unit]
```

and any function working on colors will be able to accept a bright color.

More detailed examples of this kind of inheritance can be found in the first half of [Cardelli 84b].

6.2. Bounded Universal Quantification and Subtyping

We now come to the problem of how to mix subtyping and parametric polymorphism. We have seen the usefulness of those two concepts in separate applications; and we shall now show that it is useful, and sometimes necessary, to merge them.

Let us take a simple function on records of one component:

```
value f0 = fun(x: {one: Int}) x.one
```

which can be applied to records like {one = 3, two = true}. This can be made polymorphic by:

```
value f = all[a] fun(x: {one: a}) x.one;
```

We can use f[t] on records of the form {one = y} for any y of type t, and on records like {one = y, two = true}.

The notation all[a]e allows us to express the notion that a type variable ranges over all types but does not allow us to designate type variables that range over a subset of the set of types. A general facility for specifying variables that range over arbitrary subsets of types could be realized by quantification over type sets defined by specified predicates. However we do not need this generality and can be satisfied with specifying just a particular class of subsets – namely the set of all subtypes of a given type. This may be accomplished by bounded quantification.

A type variable ranging over the set of all subtypes of a type T may be specified by bounded quantification as follows:

```
all[a ≤ T] e                -- a ranges over all subtypes of T in the scope e
```

Here is a function which accepts any record having integer component one and extracts its contents:

```
value g0 = all[a ≤ {one: Int}] fun(x: a) x.one
g0 [{one: Int, two: Bool}]({one=3, two=true})
```

Note that there is little difference between g_0 and f_0; all we have done is to move the constraint that the argument must be a subtype of {one :Int} from the fun parameter to the all parameter. We now have two ways of expressing inclusion constraints: implicitly by function parameters and explicitly by bounded quantifiers. Now that we have bounded quantifiers we could remove the other mechanism, requiring exact matching of types on parameter passing, but we shall leave it for convenience.

To express the type of g_0, we need to introduce bounded quantification in type expressions:

$$g_0 : \forall a \le \{one: Int\}.\ a \to Int$$

Now we have a way of expressing both inheritance and parametric polymorphism. Here is a new version of g_0 in which we abstract Int to any type:

```
value g = all[b] all[a ≤ {one: b}] fun(x: a) x.one
g[Int][{one: Int, two: Bool}]({one=3, two=true})
```

where all[b] e is now an abbreviation for all[b ≤ Top] e. The new function g could not be expressed by parametric polymorphism or by inheritance separately. Only their combination, achieved by bounded quantifiers, allows us to write it.

So far, bounded quantifiers have not shown any extra power, because we can rephrase g_0 as f_0 and g as f, given that we allow type inclusion on parameter passing. But bounded quantifiers are indeed more expressive, as is shown in the next example.

The need for bounded quantification arises very frequently in object-oriented programming. Suppose we have the following types and functions:

```
type Point = {x: Int, y: Int}

value moveX0 = fun(p:Point, dx:Int) p.x := p.x + dx; p
value moveX = all[P ≤ Point] fun(p:P, dx:Int) p.x := p.x + dx; p
```

It is typical in (type-free) object-oriented programming to reuse functions like moveX on objects whose type was not known when moveX was defined. If we now define:

```
type Tile = {x: Int, y:Int, hor: Int, ver:Int}
```

we may want to use moveX to move tiles, not just points. However, if we use the simpler $moveX_0$ function, it is only sound to assume that the result will be a point, even if the parameter was a tile and we allow inclusion on function arguments. Hence, we lose type information by passing a tile through the $moveX_0$ function and, for example, we cannot further extract the hor component from the result.

Bounded quantification allows us to better express input/output dependencies: the result type of moveX will be the same as its argument type, whatever subtype of Point that happens to be. Hence we can apply moveX to a tile and get a tile back without losing type information:

```
moveX[Tile]({x=0,y=0,hor=1,ver=1},1).hor
```

This shows that bounded quantification is useful even in the absence of proper parametric polymorphism to express adequately subtyping relations.

Earlier we saw that parametric polymorphism can be either explicit (by using ∀ quantifiers) or implicit (by having free type variables, implicitly quantified). We have a similar situation here, where inheritance can be either explicit, by using bounded quantifiers, or left implicit in the inclusion rules for parameter passing. In object-oriented languages, subtype parameters are generally implicit. We may consider such languages as abbreviated version of languages using bounded quantification. Thus, bounded quantification is useful not only to increase expressive power, but also to make explicit the parameter mechanisms through which inheritance is achieved.

6.3. Comparison with Other Subtyping Mechanisms

How does the above inheritance mechanisms compare with those in Simula, Smalltalk and Lisp Flavors? For many uses of inheritance the correspondence is not exact, although it can be obtained by paraphrases. Other uses of inheritance cannot be simulated, especially those which make essential use of dynamic typing. On the other hand, there are some things that can be done with bounded quantification, which are impossible in some object-oriented languages.

Record types are used to model classes and subclasses. Record types are matched by structure, and the (multiple) inheritance relations are implicit in the names and types of record components. In Simula and Smalltalk classes are matched by name, and the inheritance relations are explicit; only single inheritance is allowed. Lisp Flavors allow a form of multiple inheritance. Smalltalk's *metaclasses* cannot be emulated in the present framework.

Records are used to model class instances. Records have to be constructed explicitly (there is no *create new instance of class X* primitive), by specifying at construction time the values of the components. Hence, different records of the same record type can have different components; this gives a degree of flexibility which is not shared by Simula and Smalltalk. Simula distinguishes between functional components (operations), which must be shared by all the instances of a class, and non-functional

components (variables) which belong to instances. Simula's *virtual* procedures are a way of introducing functional components that may change in different instances of a class, but must still be uniform within subclasses of that class. Smalltalk also distinguishes between *methods*, shared by all instances of a class, and *instance variables*, local to instances. Unlike Simula's variables declared in classes, Smalltalk instance variables are *private* and cannot be directly accessed. This behavior can be easily obtained in our framework by limiting visibility of local variables via static scoping techniques.

Functional record components are used to model *methods*. As remarked in the previous paragraph, record components are conceptually bound to individual records, not to record types (although implementations can optimize this). In Simula and Smalltalk it is possible for a subclass automatically to inherit the methods of its superclass, or to redefine them. When considering multiple inheritance, this automatic way of inheriting methods creates problems in case more than one superclass defines the same method: which one should be inherited? We avoid this problem by having to create records explicitly. At record creation time one must choose explicitly which field values a particular record should have; whether it should *inherit* them by using some predefined function (or value) used in the allocation of other records, or *redefine* them, by using a new function (or value). Everything is allowed as long as the type constraints are respected.

Record field selection is used to model *message passing*. A message sent to an object with some parameters translates to the selection of a functional component of a record and its application to the parameters. This is very similar to what Simula does, while Smalltalk goes through a complex name-binding procedure to associate message names with actual methods. Simula can compute statically the precise location of a variable or operation in an instance. Smalltalk has to do a dynamic search, which can be optimized by caching recently used methods. Our field selections have intermediate complexity: because of multiple inheritance it is not possible to determine statically the precise location of a field in a record, but caching can achieve an almost constant-time access to fields, on the average, and achieves exactly constant time in programs which only use single inheritance.

Smalltalk's concept of *self*, corresponding to Simula's *this* (a class instance referring to its own methods and variables), can also be simulated without introducing any special construct. This can be done by defining a record recursively, so that an ordinary variable called *self* (though we could use a different name) refers to the record itself, recursively. Smalltalk's concept of *super* (a class instance referring to the methods of its immediate superclass) and similar constructs in Simula (*qua*) and Flavors cannot be simulated because they imply an explicit class hierarchy.

Simula has a special construct, called *inspect*, which is essentially a case statement on the class of an object. We have no way of emulating this directly; it turns out, however, that *inspect* is often used because Simula does not have variant types. Variant types in Simula are obtained by declaring all the variant cases as subclasses of an (often) dummy class and then doing an *inspect* on objects of that class. As we have variants, we just have to rephrase the relevant Simula classes and subclasses as variants and then use an ordinary *case* for discrimination.

Smalltalk and Lisp Flavors have some idioms which cannot be reproduced because they are essentially impossible to type-check statically. For example, in Flavors one can ask whether an object supports a message (although it may be possible to paraphrase some of these situations by variant types). Generally, the freedom of type-free languages is hard to match, but we have shown in previous sections that polymorphism can go a long way in achieving flexibility, and bounded quantification can extend that flexibility to inheritance situations.

6.4. Bounded Existential Quantification and Partial Abstraction

As we have done for universal quantifiers, we can modify our existential type quantifiers, restricting an existential variable to be a subtype of some type:

$$\exists a \le t.\ t'$$

We retain the notation $\exists a.\ t$ as an abbreviation for $\exists a \le \mathsf{Top}.\ t$.

Bounded existentials allow us to express *partially abstract* types: although a is abstract, we know it is a subtype of t, so it is no more abstract than t is. If t is itself an abstract type, we know that those two abstract types are in a subtype relation.

We can see this in the following example, in which we use a version of the pack construct modified for bounded existentials:

$$\mathsf{pack}\ [a \le t = t' \ \mathsf{in}\ t''] \ e$$

Suppose we have two abstract types, Point and Tile, and we want to use them and make them interact with each other. Suppose also that we want Tile to be a subtype of Point, but we do not want to know *why* the inclusion holds, because we want to use them abstractly. We can satisfy these requirements by the following definition:

```
type Tile = ∃P. ∃T ≤ P. TileWRT2[P,T]
```

Hence, there is a type P (point) such that there is a type T (tile) subtype of P which supports tile operations. More precisely:

```
type TileWRT2[P,T] =
      {mktile: (Int × Int × Int × Int) → T,
       origin: T → P,
       hor: T → Int,
       ver: T → Int
      }

type TileWRT[P] = ∃T ≤ P. TileWRT2[P,T]

type Tile = ∃P. TileWRT[P]
```

A tile package can be created as follows, where the concrete representations of points and tiles are as in the previous sections:

```
type PointRep = {x:Int,y:Int}
type TileRep = {x:Int,y:Int,hor:Int,ver:Int}

pack [P = PointRep in TileWRT[P]]
pack [T ≤ PointRep = TileRep in TileWRT2[P,T]]
      {mktile = fun(x:Int,y:Int,hor:Int,ver:Int) <x=x,y=y,hor=hor,ver=ver>,
       origin = fun(t:TileRep) t,
       hor = fun(t:TileRep) t.hor,
       ver = fun(t:TileRep) t.ver
      }
```

Note that origin returns a TileRep (a PointRep is expected), but tiles can be considered as points.

A function using abstract tiles can treat them as points, although how tiles and points are represented, and why tiles are subtypes of points, are unknown:

```
fun(tilePack:Tile)
      open tilePack as t [pointRep] [tileRep]
          let f = fun(p:pointRep) ...
          in f(t.tile(0,0,1,1))
```

In languages with both type inheritance and abstract types, it is natural to be able to extend inheritance to abstract types without having to reveal the representation of types. As we have just seen, bounded existential quantifiers can explain these situations and achieve a full integration of inheritance and abstraction.

7. Type Checking and Type Inference

In conventional typed languages, the compiler assigns a type to every expression and subexpression. However, the programmer does not have to specify the type of every subexpression of every expression: type information need only be placed at critical points in a program, and the rest is deduced from the context. This deduction process is called *type inference*. Typically, type information is given for local variables and for function arguments and results. The type of expressions and statements can then be inferred, given that the type of variables and basic constants is known.

Type inference is usually done bottom-up on expression trees. Given the type of the leaves (variables and constants) and type rules for the ways of combining expressions into bigger expressions, it is possible to deduce the type of whole expressions. For this to work it is sufficient to declare the type of newly introduced variables. Note that it may not be necessary to declare the return type of a function or the type of initialized variables.

```
fun (x:Int) x+1
let x = 0 in x+1
```

The ML language introduced a more sophisticated way of doing type inference. In ML it is not even necessary to specify the type of newly introduced variables, so that one can simply write:

```
fun (x) x+1
```

The type inference algorithm still works bottom-up. The type of a variable is initially taken to be unknown. In x+1 above, x would initially have type a, where a is a new *type variable* (a new type variable is introduced for every program variable). Then the Int operator would retroactively force a to be equivalent to Int. This instantiation of type variables is done by Robinson's unification algorithm [Robinson 65], which also takes care of propagating information across all the instances of the same variable, so that incompatible uses of the same variable are detected. An introductory exposition of polymorphic type inference can be found in [Cardelli 84a].

This inference algorithm is not limited to polymorphic languages. It could be added to any monomorphic typed language, with the restriction that at the end of type checking all the type variables should disappear. Expressions like fun (x) x would be ambiguous, and one would have to write fun (x:Int) x, for example, to disambiguate them.

The best type inference algorithm known is the one used in ML and similar languages. This amounts to saying that the best we know how to do is type inference for type systems with little existential quantification, no subtyping, and with a limited (but quite powerful) form of universal quantification. Moreover, in many extensions of the ML type system the type-checking problem has been shown to be undecidable.

Type inference reduces to type checking when there is so much type information in a program that the type inference task becomes trivial. More precisely we can talk of type checking when all the type expressions involved in checking a program are already explicitly contained in the program text, i.e., when there is no need to generate new type expressions during compilation and all one has to do is match existing type expressions.

We probably cannot hope to find fully automatic type-inference algorithms for the type system we have presented in this paper. However, the type-checking problem for this system turns out to be quite easy, given the amount of type information which has to be supplied with every program. This is probably the single most important property of this type system: it is very expressive without posing any major type-checking problem.

There is actually one problem, which is however shared by all polymorphic languages, and this has to do with type checking side-effects. Some restrictions have to be imposed to prevent violating the type system by storing and fetching polymorphic objects in memory locations. Examples can be found in [Gordon 79] and [Albano 83]. There are several known practical solutions to this problem [Damas 84] [Milner 84] which trade off flexibility with complexity of the type checker.

8. Hierarchical Classification of Type Systems

Type systems can be classified in terms of the type operators they admit. Figure 2 is a (partial) diagram of type systems ordered by *generality*. Each box in the diagram denotes a particularly clearcut type system; other type systems may fall in between. At the bottom of each box, we enumerate the type operators present in the type system (going from the bottom up, we only show the *new* operators). At the top of each box is a name for that type system and in the middle is the set of features it can model (again, going from the bottom up, we only list the *new* features). The diagram could be made more symmetrical, but it would then reflect the structure of existing classes of languages less precisely.

This is a classification of type systems, not of languages. A particular language may not fall on any particular point of this diagram, as it can have features which position it, to different degrees, at different points of the diagram. Also, existing language type systems will seldom fall exactly on one of the points we

have highlighted; more often they will have a combination of features which positions them somewhere between two or more highlighted points.

At the bottom we have simple first-order type systems, with Cartesian products, disjoint sums and first-order function spaces, which can be used to model records, variants and first-order procedures, respectively. A ° sign indicates an incomplete use of a more general type operator.

First-order type systems have evolved into higher-order type systems (on the left) and inheritance-based type systems (on the right). On the left side we could find Algol 68, a higher-order monomorphic language. On the right side we could find Simula 67, a single-inheritance language, and multiple-inheritance languages higher up (again, these allocations are not so clear-cut). These two classes of type systems are dominated by higher-order inheritance systems, as in Amber [Cardelli 85].

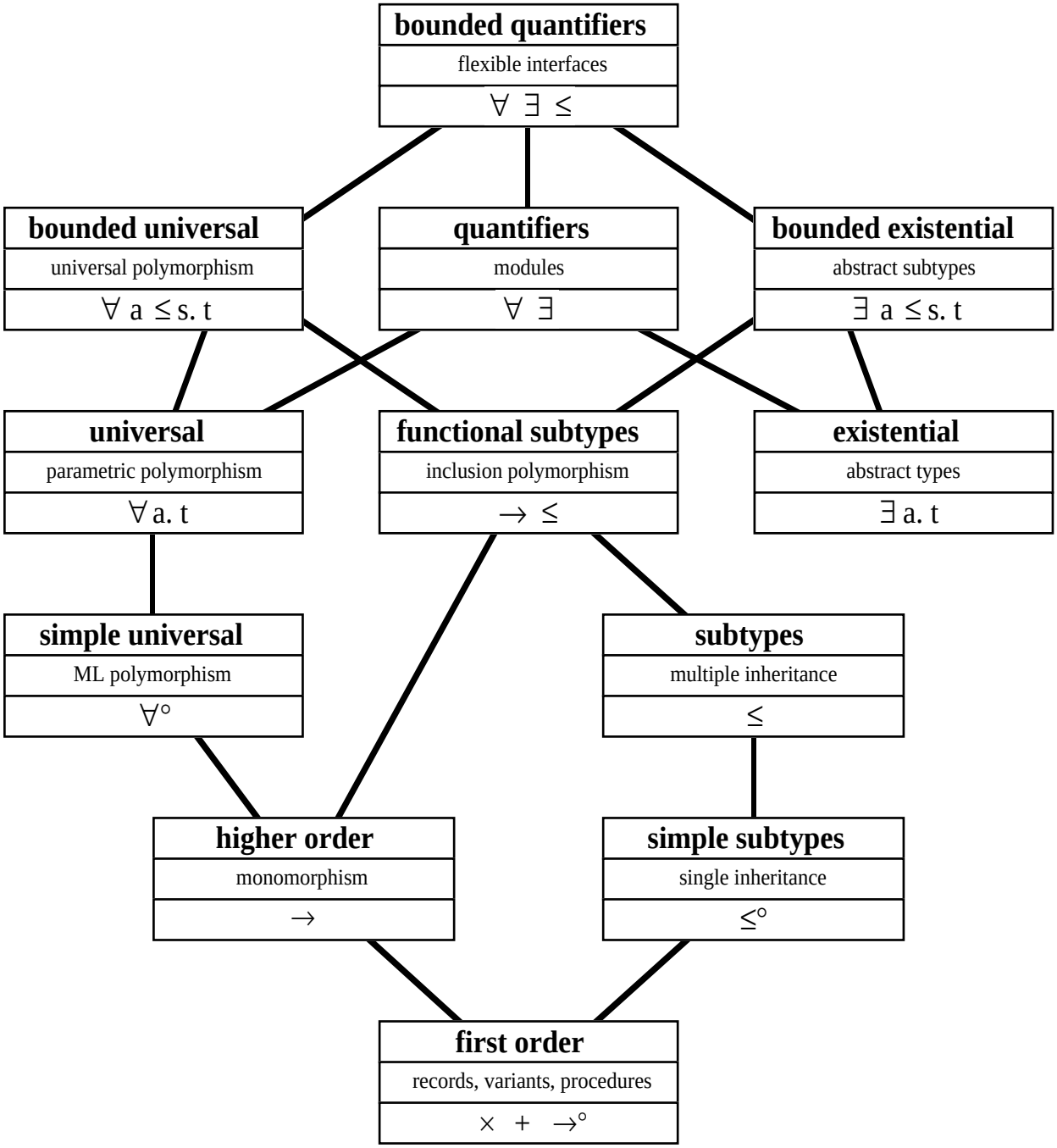

Figure 2: Classification of typ systems.

Higher-order languages have developed into parametric polymorphic languages. These can have restricted top-level universal quantification (this is Milner's type system [Milner 78], with roots in Curry [Curry 58] and Hindley [Hindley 69]) or general universal quantification (this is the Girard-Reynolds type system [Girard 71] [Reynolds 74]).

Up on the right we have type systems with type abstraction, characterized by existential quantification. Joining universal and existential quantifiers we obtain SOL's [Mitchell 85] type system, which can be used to explain basic module features.

The remaining points at the top have to do with inclusion. We have shown that the bounded universal quantifiers are needed to model object-oriented programming, and bounded existential quantifiers are needed to mix inheritance with data abstraction.

Three powerful concepts (inclusion, universal and existential quantification) are sufficient to explain most programming features. When used in full generality, they go much further than most existing languages. We have been careful to maintain the ability to type-check these features easily. However, this is not the whole picture. Many interesting type systems lie well above our diagram [Reynolds 85]. These include the Coquand and Huet theory of constructions [Coquand 85], Martin-Löf's dependent types [Martin-Löf 80], Burstall and Lampson's language Pebble [Burstall 84], and MacQueen's language DL [MacQueen 84b].

There are benefits in going even higher up: Pebble and DL have a more general treatment of parametric modules; dependent types have an almost unlimited expressive power. But there are also extra complications, which unfortunately reflect pragmatically on the complexity of type checking. The topmost point of our diagram is thus a reasonable place to stop for a while, to gain some experience, and to consider whether we are willing to accept extra complications in order to achieve extra power.

9. Conclusions

The augmented λ-calculus supports type systems with a very rich type structure in a functional framework. It can model type systems as rich, or richer, than those in real programming languages. It is sufficiently expressive to model the data abstractions of Ada and the classes of object-oriented languages. Its ability to express computations on types is strictly weaker than its ability to express computations on values.

By modeling type and module structures of real programming languages in the augmented λ-calculus, we gain an understanding of their abstract properties independent of the idiosyncrasies of programming languages in which they may be embedded. Conversely, we may view type and module structures of real programming languages as syntactically sugared versions of our augmented λ-calculus.

We started from the typed λ-calculus and augmented it with primitive types such as Int, Bool, and String and with type constructors for pairs, records, and variants.

Universal quantification was introduced to model parametric polymorphism, and existential quantification was introduced to model data abstraction. The practical application of existential quantification was demonstrated by modeling Ada packages with private data types. The usefulness of combining universal with existential abstraction was demonstrated by a generic stack example, using universal quantification to model the generic type parameter and existential quantification to model the hidden data structure.

Both universal and existential quantification become more interesting when we can restrict the domain of variation of the quantified variable. Bounded universal quantification allows more sensitive parameterization by restricting parameters to the set of all subtypes of a type. Bounded existential quantification allows more sensitive data abstraction by allowing the specification of subtyping relations between abstract types.

The insight that both subrange types of integers and subtypes defined by type inheritance are type inclusion polymorphisms extends the applicability of bounded quantification to both these cases. The case of record subtypes such as $\{a{:}T1\} \leq \{a{:}T1, b{:}T2\}$ is particularly interesting in this connection. It allows us to assert that a record type obtained by adding fields to a given record type is a subtype of that record type.

Types such as *Cars* may be modeled by record types whose fields are the set of data attributes applicable to cars. Subtypes such as *Toyotas* may be modeled by record types that include all fields of car records plus additional fields for operations applicable only to *Toyotas*. Multiple inheritance may generally be modeled by record subtypes.

Records with functional components are a very powerful mechanism for module definitions, especially when combined with mechanismd for information hiding, which are here realized by existential types. Type inclusion of records provides a paradigm for type inheritance and may be used as a basis for the design of strongly typed object-oriented languages with multiple inheritance. Although we have used

a unified language (Fun) throughout the paper, we have not presented a language design for a practical programming language. In language design there are many important issues to be solved concerning readability, easy of use, etc. which we have not directly attacked.

Fun provides a framework to classify and compare existing languages and to design new languages. We do not propose it as a programming language, as it may be clumsy in many areas, but it could be the basis of one.

Appendix: Type Inference Rules

The type system discussed in this paper can be formalized as a set of type inference rules which prescribes how to establish the type of an expression from the type of its subexpressions. These rules can be intended as the specification of a typechecking algorithm. An acceptable algorithm is one which partially agrees with these rules, in the sense that if it computes a type, that type must be derivable from the rules.

The inference rules are given in two groups: the first group is for deducing that two types are in the inclusion relation, and the second group is for deducing that an expression has a type (maybe using the first group in the process).

Type expressions are denoted by s, t and u, type variables by a and b, type constants (e.g. int) by k, expressions by e and f, variables by x, and labels by l. We identify all the type expressions which differ only because of the names of bound type variables.

Here are the rules of type inclusion $t \le s$. C is a set of *inclusion constraints* for type variables. $C.\ a \le t$ is the set C extended with the constraint that the type variable a is a subtype of the type t.

$C \vdash t \le s$ is an *assertion* meaning that from C we can infer $t \le s$. A horizontal bar is a *logic implication*: if we can infer what is above it, then we can infer what is below it.

Note: this set of rules is not complete with respect to some semantic models; some valid rules have been omitted to make typechecking easier.

{TOP} $C \vdash t \le Top$

{VAR} $C.\ a \le t \vdash a \le t$

{BAS1} $C \vdash a \le a$

{BAS2} $C \vdash k \le k$

{ARROW}
$$\frac{C \vdash s' \le s \quad C \vdash t \le t'}{C \vdash s \to t \le s' \to t'}$$

{RECD}
$$\frac{C \vdash s_1 \le t_1 \;..\; C \vdash s_n \le t_n}{C \vdash \{l_1{:}s_1, \ldots, l_n{:}s_n, \ldots, l_m{:}s_m\} \le \{l_1{:}t_1, \ldots, l_n{:}t_n\}}$$

{VART}
$$\frac{C \vdash s_1 \le t_1 \;..\; C \vdash s_n \le t_n}{C \vdash [l_1{:}s_1, \ldots, l_n{:}s_n] \le [l_1{:}t_1, \ldots, l_n{:}t_n, \ldots, l_m{:}t_m]}$$

{FORALL}
$$\frac{C.\ a \le s \vdash t \le t'}{C \vdash (\forall a \le s.t) \le (\forall a \le s.t')} \quad a \text{ not free in } C$$

{EXISTS}
$$\frac{C.\ a \le s \vdash t \le t'}{C \vdash (\exists a \le s.t) \le (\exists a \le s.t')} \quad a \text{ not free in } C$$

{TRANS}
$$\frac{C \vdash s \le t \qquad C \vdash t \le u}{C \vdash s \le u}$$

Here are the typing rules for expressions e. A is a set of type assumptions for free program variables. A. x:t is the set A extended with the assumption that variable x has type t. C,A |— e: t means that from the set of constraints C and the set of assumptions A we can infer that e has type t .

[TOP] $C,A \vdash e: Top$

[VAR] $C, A.\ x:t \vdash x: t$

[ABS] $$\frac{C,A.x:s \vdash e: t}{C,A \vdash (fun(x:s)e): s \to t}$$

[APPL] $$\frac{C,A \vdash e: s \to t \quad C,A \vdash e': s}{C,A \vdash (e\ e'): t}$$

[RECD] $$\frac{C,A \vdash e_1:t_1 \ ..\ C,A \vdash e_n:t_n}{C,A \vdash \{l_1=e_1, .. ,l_n=e_n\} : \{l_1:t_1, .. ,l_n:t_n\}}$$

[SEL] $$\frac{C,A \vdash e: \{l_1:t_1, .. ,l_n:t_n\}}{C,A \vdash e.l_i: t_i} \quad i \in 1..n$$

[VART] $$\frac{C,A \vdash e:t_i}{C,A \vdash [l_i=e]: [l_1:t_1, .. ,l_n:t_n]} \quad i \in 1..n$$

[CASE] $$\frac{C,A \vdash e: [l_1:t_1, .. ,l_n:t_n] \quad C,A \vdash f_1: t_1 \to t \ \ .. \ \ C,A \vdash f_n: t_n \to t}{C,A \vdash (\text{case } e \text{ of } l_1 \Rightarrow f_1, .. , l_n \Rightarrow f_n): t}$$

[GEN] $$\frac{C.a \le s,A \vdash e: t}{C,A \vdash all[a \le s]e : \forall a \le s.t} \quad a \text{ not free in } C,A$$

[SPEC] $$\frac{C,A \vdash e: \forall a \le s.t \quad C \vdash s' \le s}{C,A \vdash e[s']: t\{s'/a\}}$$

[PACK] $$\frac{C,A \vdash e: s\{t/a\} \quad C \vdash t \le u}{C,A \vdash pack\ [a \le u = t\ in\ s]\ e : \exists a \le u.s}$$

[OPEN] $$\frac{C,A \vdash e: \exists b \le u.s \quad C.a \le u,\ A.x:s\{a/b\} \vdash e': t}{C,A \vdash open\ e\ as\ x\ [a]\ in\ e' : t} \quad a \text{ not free in } t,C,A$$

[DEFN] $$\frac{C,A \vdash e: t\{s/b\}}{C,A \vdash e: a[s]} \quad \text{if } a[b] = t \text{ is a type definition}$$

[TRANS] $$\frac{C,A \vdash e: t \qquad C \vdash t \le u}{C,A \vdash e: u}$$

Acknowledgements

Research by Peter Wegner was supported in part by IBM, Yorktown Heights, N.Y., and in part by the Office of Naval Research under Contract No. N00014-74-C0592.

Thanks are due to Dave MacQueen for countless discussions, and to John Mitchell for insights. Also to Doug McIlroy and Malcolm Atkinson for a very thorough reading of the manuscript, and to Rishiyur Nikhil and Albert Meyer for many useful comments.

References

[Aho 85] A.V.Aho, R.Sethi, J.D.Ullman: **Compilers. Principles, techniques and tools**, Addison-Wesley 1985.

[Albano 85] A.Albano, L.Cardelli, R.Orsini: **Galileo: a strongly typed, interactive conceptual language**, *Transactions on Database Systems*, June 1985, 10(2), pp. 230-260.

[Booch 83] G. Booch, **Software Engineering with Ada**, Benjamin Cummings 1983.

[Bruce 84] K.B.Bruce, R.Meyer: **The semantics of second order polymorphic lambda calculus**, in *Sematics of Data Types*, G.Kahn, D.B.MacQueen and G.Plotkin Ed. Lecture Notes in Computer Science 173, Springer-Verlag, 1984.

[Burstall 80] R.Burstall, D.MacQueen, D.Sannella: **Hope: an experimental applicative language**, *Conference Record of the 1980 LISP Conference*, Stanford, August 1980, pp. 136-143.

[Burstall 84] R.Burstall, B.Lampson, **A kernel language for abstract data types and modules**, in *Sematics of Data Types*, G.Kahn, D.B.MacQueen and G.Plotkin Ed. Lecture Notes in Computer Science 173, Springer-Verlag, 1984.

[Cardelli 84a] L.Cardelli: **Basic polymorphic typechecking**, AT&T Bell Laboratories Computing Science Technical Report 119, 1984. Also in "Polymorphism newsletters", II.1, Jan 1984.

[Cardelli 84] L.Cardelli: **A semantics of multiple inheritance**, in *Sematics of Data Types*, G.Kahn, D.B.MacQueen and G.Plotkin Ed. Lecture Notes in Computer Science n.173, Springer-Verlag 1984.

[Cardelli 85] L.Cardelli: **Amber**, *Combinators and Functional Programming Languages, Proc. of the 13th Summer School of the LITP*, Le Val D'Ajol, Vosges (France), May 1985.

[Coquand 85] T.Coquand, G.Huet: **Constructions: a higher order proof system for mechanizing mathematics**, Technical report 401, INRIA, May 1985.

[Curry 58] H.B.Curry, R.Feys: **Combinatory logic**, North-Holland, Amsterdam, 1958.

[Damas 82] L.Damas, R.Milner: **Principal type-schemes for functional programs**, *Proc. POPL 1982*, pp.207-212.

[Damas 84] PhD Thesis, Dept of Computer Science, University of Edinburgh, Scotland.

[Demers 79] A.Demers, J.Donahue: **Revised Report on Russell**, TR79-389, Computer Science Department, Cornell University, 1979.

[D.O.D. 83] US Department of Defense: **Ada reference manual**, ANSI/MIS-STD 1815, Jan 1983.

[Fairbairn 82] J.Fairbairn: **Ponder and its type system**, Technical report No 31, Nov 82, University of Cambridge, Computer Laboratory.

[Girard 71] J-Y.Girard: **Une extension de l'interprétation de Gödel a l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types**, *Proceedings of the second Scandinavian logic symposium*, J.E.Fenstad Ed. pp. 63-92, North-Holland, 1971.

[Goldberg 83] A.Goldberg, D.Robson: **Smalltalk-80. The language and its implementation**, Addison-Wesley, 1983.

[Gordon 79] M.Gordon, R.Milner, C.Wadsworth. **Edinburgh LCF**, *Lecture Notes in Computer Science, n.78*, Springer-Verlag, 1979.

[Hendler 86] J.Hendler, P.Wegner: **Viewing object-oriented programming as an enhancement of data abstraction methodology**, *Hawaii Conference on System Sciences*, January 1986.

[Hook 84] J.G.Hook: **Understanding Russell, a first attempt**, in *Sematics of Data Types*, G.Kahn, D.B.MacQueen and G.Plotkin Ed. Lecture Notes in Computer Science 173, pp. 51-67, Springer-Verlag, 1984.

[Hindley 69] R.Hindley: **The principal type scheme of an object in combinatory logic**, *Transactions of the American Mathematical Society*, Vol. 146, Dec 1969, pp. 29-60.

[Liskov 81] B.H.Liskov: **CLU Reference Manual**, *Lecture Notes in Computer Science*, 114, Springer-Verlag, 1981.

[McCracken 84] N.McCracken: **The typechecking of programs with implicit type structure**, in *Semantics of Data Types*, Lecture Notes in Computer Science n.173, pp. 301-316, Springer-Verlag 1984.

[MacQueen 84a] D.B.MacQueen, G.D.Plotkin, R.Sethi: **An ideal model for recursive polymorphic types**, *Proc. Popl 1984*. Also to appear in *Information and Control*.

[MacQueen 84b] D.B.MacQueen: **Modules for Standard ML**, *Proc. Symposium on Lisp and Functional Programming*, Austin, Texas, August 6-8 1984, pp 198-207. ACM, New York.

[MacQueen 86] D.B.MacQueen: **Using dependent types to express modular structure**, *Proc. POPL 1986*.

[Martin-Löf 80] P.Martin-Löf, **Intuitionistic type theory**, Notes by Giovanni Sambin of a series of lectures given at the University of Padova, Italy, June 1980.

[Matthews 85] D.C.J.Matthews, **Poly manual**, Technical Report No. 63, University of Cambridge, Computer Laboratory, 1985.

[Milner 78] R.Milner: **A theory of type polymorphism in programming**, *Journal of Computer and System Science 17*, pp. 348-375, 1978.

[Milner 84] R.Milner: **A proposal for Standard ML**, *Proc. Symposium on Lisp and Functional Programming*, Austin, Texas, August 6-8 1984, pp. 184-197. ACM, New York.

[Mitchell 79] J.G.Mitchell, W.Maybury, R.Sweet: **Mesa language manual**, Xerox PARC CSL-79-3, April 1979.

[Mitchell 84] J.C.Mitchell: **Type Inference and Type Containment**, in *Semantics of Data Types*, *Lecture Notes in Computer Science 173*, 51-67, Springer-Verlag, 1984.

[Mitchell 85] J.C.Mitchell, G.D.Plotkin: **Abstract types have existential type**, *Proc. Popl 1985*.

[Reynolds 74] J.C.Reynolds: **Towards a theory of type structure**, in *Colloquium sur la programmation*, pp. 408-423, Springer-Verlag *Lecture Notes in Computer Science*, n.19, 1974.

[Reynolds 85] J.C.Reynolds: **Three approaches to type structure**, *Mathematical Foundations of Software Development*, Lecture Notes in Computer Science 185, Springer-Verlag, Berlin 1985, pp. 97-138.

[Robinson 65] J.A.Robinson: **A machine-oriented logic based on the resolution principle**, *Journal of the ACM*, vol. 12, no. 1, Jan. 1956, pp. 23-49.

[Schmidt 82] E.E.Schmidt: **Controlling large software development in a distributed environment**, Xerox PARC, CSL-82-7, Dec. 1982.

[Scott 76] D.Scott: **Data types as lattices**, *SIAM Journal of Computing*, Vol 5, No 3, September 1976, pp. 523-587.

[Solomon 78] M.Solomon: **Type Definitions with Parameters**, *Proc. POPL 1978*, Tucson, Arizona.

[Strachey 67] C.Strachey: **Fundamental concepts in programming languages**, lecture notes for the *International Summer School in Computer Programming*, Copenhagen, August 1967.

[Welsh 77] J.Welsh, W.J.Sneeringer, C.A.R.Hoare: **Ambiguities and insecurities in Pascal**, *Software Practice and Experience*, November 1977.

[Wegner 83] P.Wegner: **On the unification of data and program abstraction in Ada**, *Proc POPL 1983*.

[Weinreb 81] D.Weinreb, D.Moon: **Lisp machine manual, Fourth Edition, Chapter 20: Objects, message passing, and flavors**, Symbolics Inc., 1981.

[Wirth 83] N.Wirth: **Programming in Modula-2**, Springer-Verlag 1983.